

1. [Preface: Fast Fourier Transforms](#)
2. [Introduction: Fast Fourier Transforms](#)
3. [Multidimensional Index Mapping](#)
4. [Polynomial Description of Signals](#)
5. [The DFT as Convolution or Filtering](#)
6. [Factoring the Signal Processing Operators](#)
7. [Winograd's Short DFT Algorithms](#)
8. [DFT and FFT: An Algebraic View](#)
9. [The Cooley-Tukey Fast Fourier Transform Algorithm](#)
10. [The Prime Factor and Winograd Fourier Transform Algorithms](#)
11. [Implementing FFTs in Practice](#)
12. [Algorithms for Data with Restrictions](#)
13. [Convolution Algorithms](#)
14. [Comments: Fast Fourier Transforms](#)
15. [Conclusions: Fast Fourier Transforms](#)
16. [Appendix 1: FFT Flowgraphs](#)
17. [Appendix 2: Operation Counts for General Length FFT](#)
18. [Appendix 3: FFT Computer Programs](#)
19. [Appendix 4: Programs for Short FFTs](#)

Preface: Fast Fourier Transforms

This book focuses on the discrete Fourier transform (DFT), discrete convolution, and, particularly, the fast algorithms to calculate them. These topics have been at the center of digital signal processing since its beginning, and new results in hardware, theory and applications continue to keep them important and exciting.

As far as we can tell, Gauss was the first to propose the techniques that we now call the fast Fourier transform (FFT) for calculating the coefficients in a trigonometric expansion of an asteroid's orbit in 1805 [\[link\]](#). However, it was the seminal paper by Cooley and Tukey [\[link\]](#) in 1965 that caught the attention of the science and engineering community and, in a way, founded the discipline of digital signal processing (DSP).

The impact of the Cooley-Tukey FFT was enormous. Problems could be solved quickly that were not even considered a few years earlier. A flurry of research expanded the theory and developed excellent practical programs as well as opening new applications [\[link\]](#). In 1976, Winograd published a short paper [\[link\]](#) that set a second flurry of research in motion [\[link\]](#). This was another type of algorithm that expanded the data lengths that could be transformed efficiently and reduced the number of multiplications required. The ground work for this algorithm had been set earlier by Good [\[link\]](#) and by Rader [\[link\]](#). In 1997 Frigo and Johnson developed a program they called the FFTW (fastest Fourier transform in the west) [\[link\]](#), [\[link\]](#) which is a composite of many of ideas in other algorithms as well as new results to give a robust, very fast system for general data lengths on a variety of computer and DSP architectures. This work won the 1999 Wilkinson Prize for Numerical Software.

It is hard to overemphasize the importance of the DFT, convolution, and fast algorithms. With a history that goes back to Gauss [\[link\]](#) and a compilation of references on these topics that in 1995 resulted in over 2400 entries [\[link\]](#), the FFT may be the most important numerical algorithm in science, engineering, and applied mathematics. New theoretical results still are appearing, advances in computers and hardware continually restate the basic questions, and new applications open new areas for research. It is hoped that this book will provide the background, references, programs and

incentive to encourage further research and results in this area as well as provide tools for practical applications.

Studying the FFT is not only valuable in understanding a powerful tool, it is also a prototype or example of how algorithms can be made efficient and how a theory can be developed to define optimality. The history of this development also gives insight into the process of research where timing and serendipity play interesting roles.

Much of the material contained in this book has been collected over 40 years of teaching and research in DSP, therefore, it is difficult to attribute just where it all came from. Some comes from my earlier FFT book [\[link\]](#) which was sponsored by Texas Instruments and some from the FFT chapter in [\[link\]](#). Certainly the interaction with people like Jim Cooley and Charlie Rader was central but the work with graduate students and undergraduates was probably the most formative. I would particularly like to acknowledge Ramesh Agarwal, Howard Johnson, Mike Heideman, Henrik Sorensen, Doug Jones, Ivan Selesnick, Haitao Guo, and Gary Sitton. Interaction with my colleagues, Tom Parks, Hans Schuessler, Al Oppenheim, and Sanjit Mitra has been essential over many years. Support has come from the NSF, Texas Instruments, and the wonderful teaching and research environment at Rice University and in the IEEE Signal Processing Society.

Several chapters or sections are written by authors who have extensive experience and depth working on the particular topics. Ivan Selesnick had written several papers on the design of short FFTs to be used in the prime factor algorithm (PFA) FFT and on automatic design of these short FFTs. Markus Püschel has developed a theoretical framework for “Algebraic Signal Processing” which allows a structured generation of FFT programs and a system called “Spiral” for automatically generating algorithms specifically for an architecture. Steven Johnson along with his colleague Matteo Frigo created, developed, and now maintains the powerful FFTW system: the Fastest Fourier Transform in the West. I sincerely thank these authors for their significant contributions.

I would also like to thank Prentice Hall, Inc. who returned the copyright on [The DFT as Convolution or Filtering](#) of **Advanced Topics in Signal Processing** [\[link\]](#) around which some of this book is built. The content of

this book is in the Connexions (<http://cnx.org/content/col10550/>) repository and, therefore, is available for on-line use, **pdf** down loading, or purchase as a printed, bound physical book. I certainly want to thank Daniel Williamson, Amy Kavalewitz, and the staff of Connexions for their invaluable help. Additional FFT material can be found in Connexions, particularly content by Doug Jones [\[link\]](#), Ivan Selesnick [\[link\]](#), and Howard Johnson, [\[link\]](#). Note that this book and all the content in Connexions are copyrighted under the Creative Commons Attribution license (<http://creativecommons.org/>).

If readers find errors in any of the modules of this collection or have suggestions for improvements or additions, please email the author of the collection or module.

C. Sidney Burrus

Houston, Texas

October 20, 2008

Introduction: Fast Fourier Transforms

The development of fast algorithms usually consists of using special properties of the algorithm of interest to remove redundant or unnecessary operations of a direct implementation. Because of the periodicity, symmetries, and orthogonality of the basis functions and the special relationship with convolution, the discrete Fourier transform (DFT) has enormous capacity for improvement of its arithmetic efficiency.

There are four main approaches to formulating efficient DFT [\[link\]](#) algorithms. The first two break a DFT into multiple shorter ones. This is done in [Multidimensional Index Mapping](#) by using an index map and in [Polynomial Description of Signals](#) by polynomial reduction. The third is [Factoring the Signal Processing Operators](#) which factors the DFT operator (matrix) into sparse factors. [The DFT as Convolution or Filtering](#) develops a method which converts a prime-length DFT into cyclic convolution. Still another approach is interesting where, for certain cases, the evaluation of the DFT can be posed recursively as evaluating a DFT in terms of two half-length DFTs which are each in turn evaluated by a quarter-length DFT and so on.

The very important computational complexity theorems of Winograd are stated and briefly discussed in [Winograd's Short DFT Algorithms](#). The specific details and evaluations of the Cooley-Tukey FFT and Split-Radix FFT are given in [The Cooley-Tukey Fast Fourier Transform Algorithm](#), and PFA and WFTA are covered in [The Prime Factor and Winograd Fourier Transform Algorithms](#). A short discussion of high speed convolution is given in [Convolution Algorithms](#), both for its own importance, and its theoretical connection to the DFT. We also present the chirp, Goertzel, QFT, NTT, SR-FFT, Approx FFT, Autogen, and programs to implement some of these.

Ivan Selesnick gives a short introduction in [Winograd's Short DFT Algorithms](#) to using Winograd's techniques to give a highly structured development of short prime length FFTs and describes a program that will automatically write these programs. Markus Pueschel presents his "Algebraic Signal Processing" in [DFT and FFT: An Algebraic View](#) on describing the various FFT algorithms. And Steven Johnson describes the

FFTW (Fastest Fourier Transform in the West) in [Implementing FFTs in Practice](#)

The organization of the book represents the various approaches to understanding the FFT and to obtaining efficient computer programs. It also shows the intimate relationship between theory and implementation that can be used to real advantage. The disparity in material devoted to the various approaches represent the tastes of this author, not any intrinsic differences in value.

A fairly long list of references is given but it is impossible to be truly complete. I have referenced the work that I have used and that I am aware of. The collection of computer programs is also somewhat idiosyncratic. They are in Matlab and Fortran because that is what I have used over the years. They also are written primarily for their educational value although some are quite efficient. There is excellent content in the Connexions book by Doug Jones [\[link\]](#).

Multidimensional Index Mapping

A change of index variable or an index mapping is used to uncouple the calculations of the discrete Fourier transform (DFT). This can result in a significant reduction in the required arithmetic and the resulting algorithm is called the fast Fourier transform (FFT).

A powerful approach to the development of efficient algorithms is to break a large problem into multiple small ones. One method for doing this with both the DFT and convolution uses a linear change of index variables to map the original one-dimensional problem into a multi-dimensional problem. This approach provides a unified derivation of the Cooley-Tukey FFT, the prime factor algorithm (PFA) FFT, and the Winograd Fourier transform algorithm (WFTA) FFT. It can also be applied directly to convolution to break it down into multiple short convolutions that can be executed faster than a direct implementation. It is often easy to translate an algorithm using index mapping into an efficient program.

The basic definition of the discrete Fourier transform (DFT) is

Equation:

$$C(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}$$

where n , k , and N are integers, $j = \sqrt{-1}$, the basis functions are the N roots of unity,

Equation:

$$W_N = e^{-j2\pi/N}$$

and $k = 0, 1, 2, \dots, N - 1$.

If the N values of the transform are calculated from the N values of the data, $x(n)$, it is easily seen that N^2 complex multiplications and approximately that same number of complex additions are required. One method for reducing this required arithmetic is to use an index mapping (a

change of variables) to change the one-dimensional DFT into a two- or higher dimensional DFT. This is one of the ideas behind the very efficient Cooley-Tukey [\[link\]](#) and Winograd [\[link\]](#) algorithms. The purpose of index mapping is to change a large problem into several easier ones [\[link\]](#), [\[link\]](#). This is sometimes called the “divide and conquer” approach [\[link\]](#) but a more accurate description would be “organize and share” which explains the process of redundancy removal or reduction.

The Index Map

For a length- N sequence, the time index takes on the values

Equation:

$$n = 0, 1, 2, \dots, N - 1$$

When the length of the DFT is not prime, N can be factored as $N = N_1 N_2$ and two new independent variables can be defined over the ranges

Equation:

$$n_1 = 0, 1, 2, \dots, N_1 - 1$$

Equation:

$$n_2 = 0, 1, 2, \dots, N_2 - 1$$

A linear change of variables is defined which maps n_1 and n_2 to n and is expressed by

Equation:

$$n = ((K_1 n_1 + K_2 n_2))_N$$

where K_i are integers and the notation $((x))_N$ denotes the integer residue of x modulo N [\[link\]](#). This map defines a relation between all possible combinations of n_1 and n_2 in [\[link\]](#) and [\[link\]](#) and the values for n in [\[link\]](#). The question as to whether all of the n in [\[link\]](#) are represented, i.e.,

whether the map is one-to-one (unique), has been answered in [\[link\]](#) showing that certain integer K_i always exist such that the map in [\[link\]](#) is one-to-one. Two cases must be considered.

Case 1.

N_1 and N_2 are relatively prime, i.e., the greatest common divisor $(N_1, N_2) = 1$.

The integer map of [\[link\]](#) is one-to-one if and only if:

Equation:

$$(K_1 = aN_2) \quad \text{and/or} \quad (K_2 = bN_1) \quad \text{and} \quad (K_1, N_1) = (K_2, N_2) = 1$$

where a and b are integers.

Case 2.

N_1 and N_2 are not relatively prime, i.e., $(N_1, N_2) > 1$.

The integer map of [\[link\]](#) is one-to-one if and only if:

Equation:

$$(K_1 = aN_2) \quad \text{and} \quad (K_2 \neq bN_1) \quad \text{and} \quad (a, N_1) = (K_2, N_2) = 1$$

or

Equation:

$$(K_1 \neq aN_2) \quad \text{and} \quad (K_2 = bN_1) \quad \text{and} \quad (K_1, N_1) = (b, N_2) = 1$$

Reference [\[link\]](#) should be consulted for the details of these conditions and examples. Two classes of index maps are defined from these conditions.

Type-One Index Map:

The map of [\[link\]](#) is called a type-one map when integers a and b exist such that

Equation:

$$K_1 = aN_2 \quad \text{and} \quad K_2 = bN_1$$

Type-Two Index Map:

The map of [\[link\]](#) is called a type-two map when integers a and b exist such that

Equation:

$$K_1 = aN_2 \quad \text{or} \quad K_2 = bN_1, \quad \text{but not both.}$$

The type-one can be used **only** if the factors of N are relatively prime, but the type-two can be used whether they are relatively prime or not. Good [\[link\]](#), Thomas, and Winograd [\[link\]](#) all used the type-one map in their DFT algorithms. Cooley and Tukey [\[link\]](#) used the type-two in their algorithms, both for a fixed radix ($N = R^M$) and a mixed radix [\[link\]](#).

The frequency index is defined by a map similar to [\[link\]](#) as

Equation:

$$k = ((K_3k_1 + K_4k_2))_N$$

where the same conditions, [\[link\]](#) and [\[link\]](#), are used for determining the uniqueness of this map in terms of the integers K_3 and K_4 .

Two-dimensional arrays for the input data and its DFT are defined using these index maps to give

Equation:

$$\hat{x}(n_1, n_2) = x((K_1 n_1 + K_2 n_2))_N$$

Equation:

$$\hat{X}(k_1, k_2) = X((K_3 k_1 + K_4 k_2))_N$$

In some of the following equations, the residue reduction notation will be omitted for clarity. These changes of variables applied to the definition of the DFT given in [\[link\]](#) give

Equation:

$$C(k) = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x(n) W_N^{K_1 K_3 n_1 k_1} W_N^{K_1 K_4 n_1 k_2} W_N^{K_2 K_3 n_2 k_1} W_N^{K_2 K_4 n_2 k_2}$$

where all of the exponents are evaluated modulo N .

The amount of arithmetic required to calculate [\[link\]](#) is the same as in the direct calculation of [\[link\]](#). However, because of the special nature of the DFT, the integer constants K_i can be chosen in such a way that the calculations are “uncoupled” and the arithmetic is reduced. The requirements for this are

Equation:

$$((K_1 K_4))_N = 0 \quad \text{and/or} \quad ((K_2 K_3))_N = 0$$

When this condition and those for uniqueness in [\[link\]](#) are applied, it is found that the K_i may **always** be chosen such that one of the terms in [\[link\]](#) is zero. If the N_i are relatively prime, it is always possible to make **both** terms zero. If the N_i are not relatively prime, only one of the terms can be set to zero. When they are relatively prime, there is a choice, it is possible to either set one or both to zero. This in turn causes one or both of the center two W terms in [\[link\]](#) to become unity.

An example of the Cooley-Tukey radix-4 FFT for a length-16 DFT uses the type-two map with $K_1 = 4$, $K_2 = 1$, $K_3 = 1$, $K_4 = 4$ giving

Equation:

$$n = 4n_1 + n_2$$

Equation:

$$k = k_1 + 4k_2$$

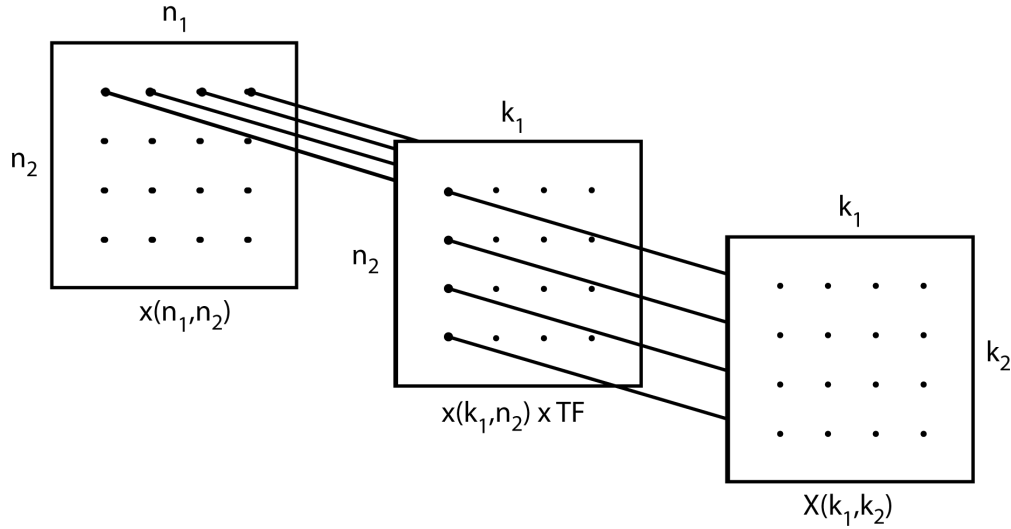
The residue reduction in [\[link\]](#) is not needed here since n does not exceed N as n_1 and n_2 take on their values. Since, in this example, the factors of N have a common factor, only one of the conditions in [\[link\]](#) can hold and, therefore, [\[link\]](#) becomes

Equation:

$$\hat{C}(k_1, k_2) = C(k) = \sum_{n_2=0}^3 \sum_{n_1=0}^3 x(n) W_4^{n_1 k_1} W_{16}^{n_2 k_1} W_4^{n_2 k_2}$$

Note the definition of W_N in [\[link\]](#) allows the simple form of $W_{16}^{K_1 K_3} = W_4$

This has the form of a two-dimensional DFT with an extra term W_{16} , called a “twiddle factor”. The inner sum over n_1 represents four length-4 DFTs, the W_{16} term represents 16 complex multiplications, and the outer sum over n_2 represents another four length-4 DFTs. This choice of the K_i “uncouples” the calculations since the first sum over n_1 for $n_2 = 0$ calculates the DFT of the first row of the data array $\hat{x}(n_1, n_2)$, and those data values are never needed in the succeeding row calculations. The row calculations are independent, and examination of the outer sum shows that the column calculations are likewise independent. This is illustrated in [\[link\]](#).



Uncoupling of the Row and Column Calculations
(Rectangles are Data Arrays)

The left 4-by-4 array is the mapped input data, the center array has the rows transformed, and the right array is the DFT array. The row DFTs and the column DFTs are independent of each other. The twiddle factors (TF) which are the center W in [\[link\]](#), are the multiplications which take place on the center array of [\[link\]](#).

This uncoupling feature reduces the amount of arithmetic required and allows the results of each row DFT to be written back over the input data locations, since that input row will not be needed again. This is called “in-place” calculation and it results in a large memory requirement savings.

An example of the type-two map used when the factors of N are relatively prime is given for $N = 15$ as

Equation:

$$n = 5n_1 + n_2$$

Equation:

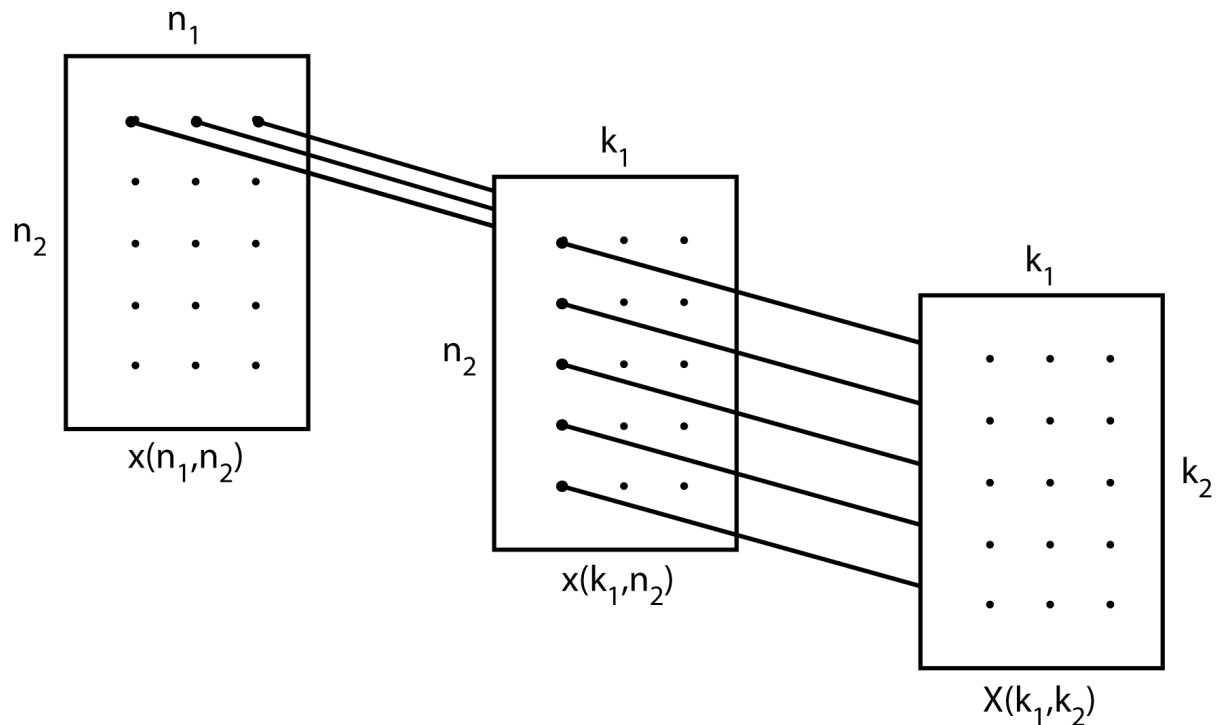
$$k = k_1 + 3k_2$$

The residue reduction is again not explicitly needed. Although the factors 3 and 5 are relatively prime, use of the type-two map sets only one of the terms in [\[link\]](#) to zero. The DFT in [\[link\]](#) becomes

Equation:

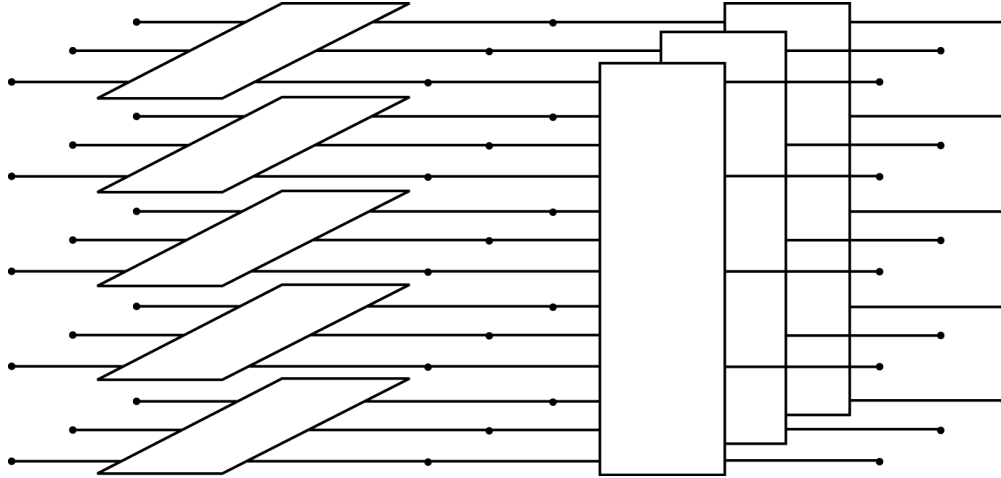
$$X = \sum_{n_2=0}^4 \sum_{n_1=0}^2 x W_3^{n_1 k_1} W_{15}^{n_2 k_1} W_5^{n_2 k_2}$$

which has the same form as [\[link\]](#), including the existence of the twiddle factors (TF). Here the inner sum is five length-3 DFTs, one for each value of k_1 . This is illustrated in [\[link\]](#) where the rectangles are the 5 by 3 data arrays and the system is called a mixed radix" FFT.



Uncoupling of the Row and Column Calculations (Rectangles are Data Arrays)

An alternate illustration is shown in [\[link\]](#) where the rectangles are the short length 3 and 5 DFTs.



Uncoupling of the Row and Column Calculations
(Rectangles are Short DFTs)

The type-one map is illustrated next on the same length-15 example. This time the situation of [\[link\]](#) with the “and” condition is used in [\[link\]](#) using an index map of

Equation:

$$n = 5n_1 + 3n_2$$

and

Equation:

$$k = 10k_1 + 6k_2$$

The residue reduction is now necessary. Since the factors of N are relatively prime and the type-one map is being used, both terms in [\[link\]](#) are

zero, and [\[link\]](#) becomes

Equation:

$$\hat{X} = \sum_{n_2=0}^4 \sum_{n_1=0}^2 \hat{x} W_3^{n_1 k_1} W_5^{n_2 k_2}$$

which is similar to [\[link\]](#), except that now the type-one map gives a pure two-dimensional DFT calculation with no TFs, and the sums can be done in either order. Figures [\[link\]](#) and [\[link\]](#) also describe this case but now there are no Twiddle Factor multiplications in the center and the resulting system is called a prime factor algorithm" (PFA).

The purpose of index mapping is to improve the arithmetic efficiency. For example a direct calculation of a length-16 DFT requires 16^2 or 256 real multiplications (recall, one complex multiplication requires 4 real multiplications and 2 real additions) and an uncoupled version requires 144. A direct calculation of a length-15 DFT requires 225 multiplications but with a type-two map only 135 and with a type-one map, 120. Recall one complex multiplication requires four real multiplications and two real additions.

Algorithms of practical interest use short DFT's that require fewer than N^2 multiplications. For example, length-4 DFTs require no multiplications and, therefore, for the length-16 DFT, only the TFs must be calculated. That calculation uses 16 multiplications, many fewer than the 256 or 144 required for the direct or uncoupled calculation.

The concept of using an index map can also be applied to convolution to convert a length $N = N_1 N_2$ one-dimensional cyclic convolution into a N_1 by N_2 two-dimensional cyclic convolution [\[link\]](#), [\[link\]](#). There is no savings of arithmetic from the mapping alone as there is with the DFT, but savings can be obtained by using special short algorithms along each dimension. This is discussed in [Algorithms for Data with Restrictions](#) .

In-Place Calculation of the DFT and Scrambling

Because use of both the type-one and two index maps uncouples the calculations of the rows and columns of the data array, the results of each short length N_i DFT can be written back over the data as it will not be needed again after that particular row or column is transformed. This is easily seen from Figures [\[link\]](#), [\[link\]](#), and [\[link\]](#) where the DFT of the first row of $x(n_1, n_2)$ can be put back over the data rather than written into a new array. After all the calculations are finished, the total DFT is in the array of the original data. This gives a significant memory savings over using a separate array for the output.

Unfortunately, the use of in-place calculations results in the order of the DFT values being permuted or scrambled. This is because the data is indexed according to the input map [\[link\]](#) and the results are put into the same locations rather than the locations dictated by the output map [\[link\]](#). For example with a length-8 radix-2 FFT, the input index map is

Equation:

$$n = 4n_1 + 2n_2 + n_3$$

which to satisfy [\[link\]](#) requires an output map of

Equation:

$$k = k_1 + 2k_2 + 4k_3$$

The in-place calculations will place the DFT results in the locations of the input map and these should be reordered or unscrambled into the locations given by the output map. Examination of these two maps shows the scrambled output to be in a “bit reversed” order.

For certain applications, this scrambled output order is not important, but for many applications, the order must be unscrambled before the DFT can be considered complete. Because the radix of the radix-2 FFT is the same as the base of the binary number representation, the correct address for any term is found by reversing the binary bits of the address. The part of most FFT programs that does this reordering is called a bit-reversed counter.

Examples of various unscramblers are found in [\[link\]](#), [\[link\]](#) and in the appendices.

The development here uses the input map and the resulting algorithm is called “decimation-in-frequency”. If the output rather than the input map is used to derive the FFT algorithm so the correct output order is obtained, the input order must be scrambled so that its values are in locations specified by the output map rather than the input map. This algorithm is called “decimation-in-time”. The scrambling is the same bit-reverse counting as before, but it precedes the FFT algorithm in this case. The same process of a post-unscrambler or pre-scrambler occurs for the in-place calculations with the type-one maps. Details can be found in [\[link\]](#), [\[link\]](#). It is possible to do the unscrambling while calculating the FFT and to avoid a separate unscrambler. This is done for the Cooley-Tukey FFT in [\[link\]](#) and for the PFA in [\[link\]](#), [\[link\]](#), [\[link\]](#).

If a radix-2 FFT is used, the unscrambler is a bit-reversed counter. If a radix-4 FFT is used, the unscrambler is a base-4 reversed counter, and similarly for radix-8 and others. However, if for the radix-4 FFT, the short length-4 DFTs (butterflies) have their outputs in bit-reversed order, the output of the total radix-4 FFT will be in bit-reversed order, not base-4 reversed order. This means any radix- 2^n FFT can use the same radix-2 bit-reversed counter as an unscrambler if the proper butterflies are used.

Efficiencies Resulting from Index Mapping with the DFT

In this section the reductions in arithmetic in the DFT that result from the index mapping alone will be examined. In practical algorithms several methods are always combined, but it is helpful in understanding the effects of a particular method to study it alone.

The most general form of an uncoupled two-dimensional DFT is given by **Equation:**

$$X(k_1, k_2) = \sum_{n_2=0}^{N_2-1} \left\{ \sum_{n_1=0}^{N_1-1} x(n_1, n_2) f_1(n_1, n_2, k_1) \right\} f_2(n_2, k_1, k_2)$$

where the inner sum calculates N_2 length- N_1 DFT's and, if for a type-two map, the effects of the TFs. If the number of arithmetic operations for a length- N DFT is denoted by $F(N)$, the number of operations for this inner sum is $F = N_2 F(N_1)$. The outer sum which gives N_1 length- N_2 DFT's requires $N_1 F(N_2)$ operations. The total number of arithmetic operations is then

Equation:

$$F = N_2 F(N_1) + N_1 F(N_2)$$

The first question to be considered is for a fixed length N , what is the optimal relation of N_1 and N_2 in the sense of minimizing the required amount of arithmetic. To answer this question, N_1 and N_2 are temporarily assumed to be real variables rather than integers. If the short length- N_i DFT's in [\[link\]](#) and any TF multiplications are assumed to require N_i^2 operations, i.e. $F(N_i) = N_i^2$, ["Efficiencies Resulting from Index Mapping with the DFT"](#) becomes

Equation:

$$F = N_2 N_1^2 + N_1 N_2^2 = N(N_1 + N_2) = N(N_1 + N N_1^{-1})$$

To find the minimum of F over N_1 , the derivative of F with respect to N_1 is set to zero (temporarily assuming the variables to be continuous) and the result requires $N_1 = N_2$.

Equation:

$$dF/dN_1 = 0 \quad \Rightarrow \quad N_1 = N_2$$

This result is also easily seen from the symmetry of N_1 and N_2 in $N = N_1 N_2$. If a more general model of the arithmetic complexity of the short DFT's is used, the same result is obtained, but a closer examination must be made to assure that $N_1 = N_2$ is a global minimum.

If only the effects of the index mapping are to be considered, then the $F(N) = N^2$ model is used and [\[link\]](#) states that the two factors should be

equal. If there are M factors, a similar reasoning shows that all M factors should be equal. For the sequence of length

Equation:

$$N = R^M$$

there are now M length- R DFT's and, since the factors are all equal, the index map must be type two. This means there must be twiddle factors.

In order to simplify the analysis, only the number of multiplications will be considered. If the number of multiplications for a length- R DFT is $F(R)$, then the formula for operation counts in [\[link\]](#) generalizes to

Equation:

$$F = N \sum_{i=1}^M F(N_i)/N_i = NMF(R)/R$$

for $N_i = R$

Equation:

$$F = N \ln R(N)F(R)/R = (N \ln N)(F(R)/(R \ln R))$$

This is a very important formula which was derived by Cooley and Tukey in their famous paper [\[link\]](#) on the FFT. It states that for a given R which is called the radix, the number of multiplications (and additions) is proportional to $N \ln N$. It also shows the relation to the value of the radix, R .

In order to get some idea of the "best" radix, the number of multiplications to compute a length- R DFT is assumed to be $F(R) = R^x$. If this is used with [\[link\]](#), the optimal R can be found.

Equation:

$$dF/dR = 0 \quad \Rightarrow \quad R = e^{1/(x-1)}$$

For $x = 2$ this gives $R = e$, with the closest integer being three.

The result of this analysis states that if no other arithmetic saving methods other than index mapping are used, and if the length- R DFT's plus TFs require $F = R^2$ multiplications, the optimal algorithm requires

Equation:

$$F = 3N \log_3 N$$

multiplications for a length $N = 3^M$ DFT. Compare this with N^2 for a direct calculation and the improvement is obvious.

While this is an interesting result from the analysis of the effects of index mapping alone, in practice, index mapping is almost always used in conjunction with special algorithms for the short length- N_i DFT's in [\[link\]](#). For example, if $R = 2$ or 4, there are no multiplications required for the short DFT's. Only the TFs require multiplications. Winograd (see [Winograd's Short DFT Algorithms](#)) has derived some algorithms for short DFT's that require $O(N)$ multiplications. This means that $F(N_i) = KN_i$ and the operation count F in ["Efficiencies Resulting from Index Mapping with the DFT"](#) is independent of N_i . Therefore, the derivative of F is zero for all N_i . Obviously, these particular cases must be examined.

The FFT as a Recursive Evaluation of the DFT

It is possible to formulate the DFT so a length- N DFT can be calculated in terms of two length- $(N/2)$ DFTs. And, if $N = 2^M$, each of those length- $(N/2)$ DFTs can be found in terms of length- $(N/4)$ DFTs. This allows the DFT to be calculated by a recursive algorithm with M recursions, giving the familiar order $N \log(N)$ arithmetic complexity.

Calculate the even indexed DFT values from [\[link\]](#) by:

Equation:

$$C(2k) = \sum_{n=0}^{N-1} x(n) W_N^{2nk} = \sum_{n=0}^{N-1} x(n) W_{N/2}^{nk}$$

Equation:

$$C(2k) = \sum_{n=0}^{N/2-1} x(n) W_N^{2nk} + \sum_{n=N/2}^{N-1} x(n) W_{N/2}^{nk}$$

Equation:

$$C(2k) = \sum_{n=0}^{N/2-1} \{x(n) + x(n + N/2)\} W_{N/2}^{nk}$$

and a similar argument gives the odd indexed values as:

Equation:

$$C(2k+1) = \sum_{n=0}^{N/2-1} \{x(n) - x(n + N/2)\} W_N^n W_{N/2}^{nk}$$

Together, these are recursive DFT formulas expressing the length-N DFT of $x(n)$ in terms of length-N/2 DFTs:

Equation:

$$C(2k) = \text{DFT}_{N/2} \{x(n) + x(n + N/2)\}$$

Equation:

$$C(2k+1) = \text{DFT}_{N/2} \{[x(n) - x(n + N/2)]W_N^n\}$$

This is a “decimation-in-frequency” (DIF) version since it gives samples of the frequency domain representation in terms of blocks of the time domain signal.

A recursive Matlab program which implements this is given by:

```
function c = dftr2(x)
```



```

% Recursive Decimation-in-Frequency FFT
algorithm, csb 8/21/07
L = length(x);
if L > 1
    L2 = L/2;
    TF = exp(-j*2*pi/L).^[0:L2-1];
    c1 = dftr2( x(1:L2) + x(L2+1:L));
    c2 = dftr2((x(1:L2) - x(L2+1:L)).*TF);
    cc = [c1';c2'];
    c = cc(:);
else
    c = x;
end
DIF Recursive FFT for  $N = 2^M$ 

```

A DIT version can be derived in the form:

Equation:

$$C(k) = \text{DFT}_{N/2} \{x(2n)\} + W_N^k \text{DFT}_{N/2} \{x(2n+1)\}$$

Equation:

$$C(k + N/2) = \text{DFT}_{N/2} \{x(2n)\} - W_N^k \text{DFT}_{N/2} \{x(2n+1)\}$$

which gives blocks of the frequency domain from samples of the signal.

A recursive Matlab program which implements this is given by:

```

function c = dftr(x)
% Recursive Decimation-in-Time FFT algorithm, csb
L = length(x);
if L > 1
    L2 = L/2;
    ce = dftr(x(1:2:L-1));
    co = dftr(x(2:2:L));
    TF = exp(-j*2*pi/L).^[0:L2-1];
    c1 = TF.*co;

```

```

        c  = [(ce+c1), (ce-c1)];
else
        c  = x;
end

```

DIT Recursive FFT for $N = 2^M$

Similar recursive expressions can be developed for other radices and algorithms. Most recursive programs do not execute as efficiently as looped or straight code, but some can be very efficient, e.g. parts of the FFTW.

Note a length- 2^M sequence will require M recursions, each of which will require $N/2$ multiplications. This give the $N \log (N)$ formula that the other approaches also derive.

Polynomial Description of Signals

Polynomials are important in digital signal processing because calculating the DFT can be viewed as a polynomial evaluation problem and convolution can be viewed as polynomial multiplication [\[link\]](#), [\[link\]](#). Indeed, this is the basis for the important results of Winograd discussed in [Winograd's Short DFT Algorithms](#). A length- N signal $x(n)$ will be represented by an $N - 1$ degree polynomial $X(s)$ defined by **Equation:**

$$X(s) = \sum_{n=0}^{N-1} x(n) s^n$$

This polynomial $X(s)$ is a single entity with the coefficients being the values of $x(n)$. It is somewhat similar to the use of matrix or vector notation to efficiently represent signals which allows use of new mathematical tools.

The convolution of two finite length sequences, $x(n)$ and $h(n)$, gives an output sequence defined by

Equation:

$$y(n) = \sum_{k=0}^{N-1} x(k) h(n - k)$$

$n = 0, 1, 2, \dots, 2N - 1$ where $h(k) = 0$ for $k < 0$. This is exactly the same operation as calculating the coefficients when multiplying two polynomials. Equation [\[link\]](#) is the same as

Equation:

$$Y(s) = X(s) H(s)$$

In fact, convolution of number sequences, multiplication of polynomials, and the multiplication of integers (except for the carry operation) are all the same operations. To obtain cyclic convolution, where the indices in [\[link\]](#) are all evaluated modulo N , the polynomial multiplication in [\[link\]](#) is done modulo the polynomial $P(s) = s^N - 1$. This is seen by noting that $N = 0 \bmod N$, therefore, $s^N = 1$ and the polynomial modulus is $s^N - 1$.

Polynomial Reduction and the Chinese Remainder Theorem

Residue reduction of one polynomial modulo another is defined similarly to residue reduction for integers. A polynomial $F(s)$ has a residue polynomial $R(s)$ modulo $P(s)$ if, for a given $F(s)$ and $P(s)$, a $Q(s)$ and $R(s)$ exist such that

Equation:

$$F(s) = Q(s)P(s) + R(s)$$

with $\text{degree}\{R(s)\} < \text{degree}\{P(s)\}$. The notation that will be used is

Equation:

$$R(s) = ((F(s)))_{P(s)}$$

For example,

Equation:

$$(s + 1) = ((s^4 + s^3 - s - 1))_{(s^2-1)}$$

The concepts of factoring a polynomial and of primeness are an extension of these ideas for integers. For a given allowed set of coefficients (values of $x(n)$), any polynomial has a unique factored representation

Equation:

$$F(s) = \prod_{i=1}^M F_i(s)^{k_i}$$

where the $F_i(s)$ are relatively prime. This is analogous to the fundamental theorem of arithmetic.

There is a very useful operation that is an extension of the integer Chinese Remainder Theorem (CRT) which says that if the modulus polynomial can be factored into relatively prime factors

Equation:

$$P(s) = P_1(s) P_2(s)$$

then there exist two polynomials, $K_1(s)$ and $K_2(s)$, such that any polynomial $F(s)$ can be recovered from its residues by

Equation:

$$F(s) = K_1(s)F_1(s) + K_2(s)F_2(s) \mod P(s)$$

where F_1 and F_2 are the residues given by

Equation:

$$F_1(s) = ((F(s)))_{P_1(s)}$$

and

Equation:

$$F_2(s) = ((F(s)))_{P_2(s)}$$

if the order of $F(s)$ is less than $P(s)$. This generalizes to any number of relatively prime factors of $P(s)$ and can be viewed as a means of representing $F(s)$ by several lower degree polynomials, $F_i(s)$.

This decomposition of $F(s)$ into lower degree polynomials is the process used to break a DFT or convolution into several simple problems which are solved and then recombined using the CRT of [\[link\]](#). This is another form of the “divide and conquer” or “organize and share” approach similar to the index mappings in [Multidimensional Index Mapping](#).

One useful property of the CRT is for convolution. If cyclic convolution of $x(n)$ and $h(n)$ is expressed in terms of polynomials by

Equation:

$$Y(s) = H(s)X(s) \mod P(s)$$

where $P(s) = s^N - 1$, and if $P(s)$ is factored into two relatively prime factors $P = P_1P_2$, using residue reduction of $H(s)$ and $X(s)$ modulo P_1 and P_2 , the lower degree residue polynomials can be multiplied and the results recombined with the CRT. This is done by

Equation:

$$Y(s) = ((K_1H_1X_1 + K_2H_2X_2))_P$$

where

Equation:

$$H_1 = ((H))_{P_1}, \quad X_1 = ((X))_{P_1}, \quad H_2 = ((H))_{P_2}, \quad X_2 = ((X))_{P_2}$$

and K_1 and K_2 are the CRT coefficient polynomials from [\[link\]](#). This allows two shorter convolutions to replace one longer one.

Another property of residue reduction that is useful in DFT calculation is polynomial evaluation. To evaluate $F(s)$ at $s = x$, $F(s)$ is reduced modulo $s - x$.

Equation:

$$F(x) = ((F(s)))_{s-x}$$

This is easily seen from the definition in [\[link\]](#)

Equation:

$$F(s) = Q(s)(s - x) + R(s)$$

Evaluating $s = x$ gives $R(s) = F(x)$ which is a constant. For the DFT this becomes

Equation:

$$C(k) = ((X(s)))_{s-W^k}$$

Details of the polynomial algebra useful in digital signal processing can be found in [\[link\]](#), [\[link\]](#), [\[link\]](#).

The DFT as a Polynomial Evaluation

The Z-transform of a number sequence $x(n)$ is defined as

Equation:

$$X(z) = \sum_{n=0}^{\infty} x(n) z^{-n}$$

which is the same as the polynomial description in [\[link\]](#) but with a negative exponent. For a finite length-N sequence [\[link\]](#) becomes

Equation:

$$X(z) = \sum_{n=0}^{N-1} x(n) z^{-n}$$

Equation:

$$X(z) = x(0) + x(1)z^{-1} + x(2)z^{-2} + \dots + x(N-1)z^{-(N-1)}$$

This $N - 1$ order polynomial takes on the values of the DFT of $x(n)$ when evaluated at

Equation:

$$z = e^{j2\pi k/N}$$

which gives

Equation:

$$C(k) = X(z)|_{z=e^{j2\pi k/N}} = \sum_{n=0}^{N-1} x(n) e^{-j2\pi nk/N}$$

In terms of the positive exponent polynomial from [\[link\]](#), the DFT is

Equation:

$$C(k) = X(s)|_{s=W^k}$$

where

Equation:

$$W = e^{-j2\pi/N}$$

is an N^{th} root of unity (raising W to the N^{th} power gives one). The N values of the DFT are found from $X(s)$ evaluated at the N N^{th} roots of unity which are equally spaced around the unit circle in the complex s plane.

One method of evaluating $X(z)$ is the so-called Horner's rule or nested evaluation. When expressed as a recursive calculation, Horner's rule becomes the Goertzel algorithm which has some computational advantages especially when only a few values of the DFT are needed. The details and programs can be found in [\[link\]](#), [\[link\]](#)

and [The DFT as Convolution or Filtering: Goertzel's Algorithm \(or A Better DFT Algorithm\)](#).

Another method for evaluating $X(s)$ is the residue reduction modulo $(s - W^k)$ as shown in [\[link\]](#). Each evaluation requires N multiplications and therefore, N^2 multiplications for the N values of $C(k)$.

Equation:

$$C(k) = ((X(s)))_{(s-W^k)}$$

A considerable reduction in required arithmetic can be achieved if some operations can be shared between the reductions for different values of k . This is done by carrying out the residue reduction in stages that can be shared rather than done in one step for each k in [\[link\]](#).

The N values of the DFT are values of $X(s)$ evaluated at s equal to the N roots of the polynomial $P(s) = s^N - 1$ which are W^k . First, assuming N is even, factor $P(s)$ as

Equation:

$$P(s) = (s^N - 1) = P_1(s) P_2(s) = (s^{N/2} - 1) (s^{N/2} + 1)$$

$X(s)$ is reduced modulo these two factors to give two residue polynomials, $X_1(s)$ and $X_2(s)$. This process is repeated by factoring P_1 and further reducing X_1 then factoring P_2 and reducing X_2 . This is continued until the factors are of first degree which gives the desired DFT values as in [\[link\]](#). This is illustrated for a length-8 DFT. The polynomial whose roots are W^k , factors as

Equation:

$$P(s) = s^8 - 1$$

Equation:

$$= [s^4 - 1] [s^4 + 1]$$

Equation:

$$= [(s^2 - 1) (s^2 + 1)] [(s^2 - j) (s^2 + j)]$$

Equation:

$$= [(s-1)(s+1)(s-j)(s+j)][(s-a)(s+a)(s-ja)(s+ja)]$$

where $a^2 = j$. Reducing $X(s)$ by the first factoring gives two third degree polynomials

Equation:

$$X(s) = x_0 + x_1s + x_2s^2 + \dots + x_7s^7$$

gives the residue polynomials

Equation:

$$X_1(s) = ((X(s)))_{(s^4-1)} = (x_0 + x_4) + (x_1 + x_5)s + (x_2 + x_6)s^2 + (x_3 + x_7)s^3$$

Equation:

$$X_2(s) = ((X(s)))_{(s^4+1)} = (x_0 - x_4) + (x_1 - x_5)s + (x_2 - x_6)s^2 + (x_3 - x_7)s^3$$

Two more levels of reduction are carried out to finally give the DFT. Close examination shows the resulting algorithm to be the decimation-in-frequency radix-2 Cooley-Tukey FFT [\[link\]](#), [\[link\]](#). Martens [\[link\]](#) has used this approach to derive an efficient DFT algorithm.

Other algorithms and types of FFT can be developed using polynomial representations and some are presented in the generalization in [DFT and FFT: An Algebraic View](#).

The DFT as Convolution or Filtering

A major application of the FFT is fast convolution or fast filtering where the DFT of the signal is multiplied term-by-term by the DFT of the impulse (helps to be doing finite impulse response (FIR) filtering) and the time-domain output is obtained by taking the inverse DFT of that product. What is less well-known is the DFT can be calculated by convolution. There are several different approaches to this, each with different application.

Rader's Conversion of the DFT into Convolution

In this section a method quite different from the index mapping or polynomial evaluation is developed. Rather than dealing with the DFT directly, it is converted into a cyclic convolution which must then be carried out by some efficient means. Those means will be covered later, but here the conversion will be explained. This method requires use of some number theory, which can be found in an accessible form in [\[link\]](#) or [\[link\]](#) and is easy enough to verify on one's own. A good general reference on number theory is [\[link\]](#).

The DFT and cyclic convolution are defined by

Equation:

$$C(k) = \sum_{n=0}^{N-1} x(n) W^{nk}$$

Equation:

$$y(k) = \sum_{n=0}^{N-1} x(n) h(k - n)$$

For both, the indices are evaluated modulo N . In order to convert the DFT in [\[link\]](#) into the cyclic convolution of [\[link\]](#), the nk product must be changed to the $k - n$ difference. With real numbers, this can be done with logarithms, but it is more complicated when working in a finite set of

integers modulo N . From number theory [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), it can be shown that if the modulus is a prime number, a base (called a primitive root) exists such that a form of integer logarithm can be defined. This is stated in the following way. If N is a prime number, a number r called a primitive roots exists such that the integer equation

Equation:

$$n = ((r^m))_N$$

creates a unique, one-to-one map of the $N - 1$ member set $m = \{0, \dots, N - 2\}$ and the $N - 1$ member set $n = \{1, \dots, N - 1\}$. This is because the multiplicative group of integers modulo a prime, p , is isomorphic to the additive group of integers modulo $(p - 1)$ and is illustrated for $N = 5$ below.

r	m=	0	1	2	3	4	5	6	7
1		1	1	1	1	1	1	1	1
2		1	2	4	3	1	2	4	3
3		1	3	4	2	1	3	4	2
4		1	4	1	4	1	4	1	4
5		*	0	0	0	*	0	0	0
6		1	1	1	1	1	1	1	1

Table of Integers $n = ((r^m))$ modulo 5, [* not defined]

[\[link\]](#) is an array of values of r^m modulo N and it is easy to see that there are two primitive roots, 2 and 3, and [\[link\]](#) defines a permutation of the integers n from the integers m (except for zero). [\[link\]](#) and a primitive root (usually chosen to be the smallest of those that exist) can be used to convert the DFT in [\[link\]](#) to the convolution in [\[link\]](#). Since [\[link\]](#) cannot give a zero, a new length- $(N-1)$ data sequence is defined from $x(n)$ by removing the term with index zero. Let

Equation:

$$n = r^{-m}$$

and

Equation:

$$k = r^s$$

where the term with the negative exponent (the inverse) is defined as the integer that satisfies

Equation:

$$\left((r^{-m} r^m) \right)_N = 1$$

If N is a prime number, r^{-m} always exists. For example, $\left((2^{-1}) \right)_5 = 3$.

[\[link\]](#) now becomes

Equation:

$$C(r^s) = \sum_{m=0}^{N-2} x(r^{-m}) W^{r^{-m} r^s} + x(0),$$

for $s = 0, 1, \dots, N-2$, and

Equation:

$$C(0) = \sum_{n=0}^{N-1} x(n)$$

New functions are defined, which are simply a permutation in the order of the original functions, as

Equation:

$$x'(m) = x(r^{-m}), \quad C'(s) = C(r^s), \quad W'(n) = W^{r^n}$$

[\[link\]](#) then becomes

Equation:

$$C'(s) = \sum_{m=0}^{N-2} x'(m) W'(s-m) + x(0)$$

which is cyclic convolution of length N-1 (plus $x(0)$) and is denoted as

Equation:

$$C'(k) = x'(k) * W'(k) + x(0)$$

Applying this change of variables (use of logarithms) to the DFT can best be illustrated from the matrix formulation of the DFT. [\[link\]](#) is written for a length-5 DFT as

Equation:

$$\begin{bmatrix} C(0) \\ C(1) \\ C(2) \\ C(3) \\ C(4) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 \\ 0 & 2 & 4 & 1 & 3 \\ 0 & 3 & 1 & 4 & 2 \\ 0 & 4 & 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \end{bmatrix}$$

where the square matrix should contain the terms of W^{nk} but for clarity, only the exponents nk are shown. Separating the $x(0)$ term, applying the mapping of [\[link\]](#), and using the primitive roots $r = 2$ (and $r^{-1} = 3$) gives **Equation:**

$$\begin{bmatrix} C(1) \\ C(2) \\ C(4) \\ C(3) \end{bmatrix} = \begin{bmatrix} 1 & 3 & 4 & 2 \\ 2 & 1 & 3 & 4 \\ 4 & 2 & 1 & 3 \\ 3 & 4 & 2 & 1 \end{bmatrix} \begin{bmatrix} x(1) \\ x(3) \\ x(4) \\ x(2) \end{bmatrix} + \begin{bmatrix} x(0) \\ x(0) \\ x(0) \\ x(0) \end{bmatrix}$$

and

Equation:

$$C(0) = x(0) + x(1) + x(2) + x(3) + x(4)$$

which can be seen to be a reordering of the structure in [\[link\]](#). This is in the form of cyclic convolution as indicated in [\[link\]](#). Rader first showed this in 1968 [\[link\]](#), stating that a prime length-N DFT could be converted into a length-(N-1) cyclic convolution of a permutation of the data with a permutation of the W's. He also stated that a slightly more complicated version of the same idea would work for a DFT with a length equal to an odd prime to a power. The details of that theory can be found in [\[link\]](#), [\[link\]](#).

Until 1976, this conversion approach received little attention since it seemed to offer few advantages. It has specialized applications in calculating the DFT if the cyclic convolution is done by distributed arithmetic table look-up [\[link\]](#) or by use of number theoretic transforms [\[link\]](#), [\[link\]](#), [\[link\]](#). It and the Goertzel algorithm [\[link\]](#), [\[link\]](#) are efficient when only a few DFT values need to be calculated. It may also have advantages when used with pipelined or vector hardware designed for fast inner products. One example is the TMS320 signal processing microprocessor which is pipelined for inner products. The general use of this scheme emerged when new fast cyclic convolution algorithms were developed by Winograd [\[link\]](#).

The Chirp Z-Transform (or Bluestein's Algorithm)

The DFT of $x(n)$ evaluates the Z-transform of $x(n)$ on N equally spaced points on the unit circle in the z plane. Using a nonlinear change of variables, one can create a structure which is equivalent to modulation and filtering $x(n)$ by a “chirp” signal. [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#).

The mathematical identity $(k - n)^2 = k^2 - 2kn + n^2$ gives

Equation:

$$nk = \left(n^2 - (k - n)^2 + k^2 \right) / 2$$

which substituted into the definition of the DFT in [Multidimensional Index Mapping: Equation 1](#) gives

Equation:

$$C(k) = \left\{ \sum_{n=0}^{N-1} \left[x(n) W^{n^2/2} \right] W^{-(k-n)^2/2} \right\} W^{k^2/2}$$

This equation can be interpreted as first multiplying (modulating) the data $x(n)$ by a chirp sequence $(W^{n^2/2})$, then convolving (filtering) it, then finally multiplying the filter output by the chirp sequence to give the DFT.

Define the chirp sequence or signal as $h(n) = W^{n^2/2}$ which is called a chirp because the squared exponent gives a sinusoid with changing frequency. Using this definition, [\[link\]](#) becomes

Equation:

$$C(n) = \{ [x(n) h(n)] * h^{-1} \} h(n)$$

We know that convolution can be carried out by multiplying the DFTs of the signals, here we see that evaluation of the DFT can be carried out by convolution. Indeed, the convolution represented by $*$ in [\[link\]](#) can be

carried out by DFTs (actually FFTs) of a larger length. This allows a prime length DFT to be calculated by a very efficient length- 2^M FFT. This becomes practical for large N when a particular non-composite (or N with few factors) length is required.

As developed here, the chirp z-transform evaluates the z-transform at equally spaced points on the unit circle. A slight modification allows evaluation on a spiral and in segments [\[link\]](#), [\[link\]](#) and allows savings with only some input values are nonzero or when only some output values are needed. The story of the development of this transform is given in [\[link\]](#).

Two Matlab programs to calculate an arbitrary length DFT using the chirp z-transform is shown in [\[link\]](#).

```
function y = chirpc(x);
% function y = chirpc(x)
% computes an arbitrary-length DFT with the
% chirp z-transform algorithm.  csb.  6/12/91
%
N = length(x);  n = 0:N-1;          %Sequence length
W = exp(-j*pi*n.*n/N);              %Chirp signal
xw = x.*W;                          %Modulate with
chirp
WW = [conj(W(N:-1:2)), conj(W)];    %Construct filter
y = conv(WW,xw);                    %Convolve w filter
y = y(N:2*N-1).*W;                  %Demodulate w
chirp
```

```
function y = chirp(x);
% function y = chirp(x)
% computes an arbitrary-length Discrete Fourier
Transform (DFT)
% with the chirp z transform algorithm. The linear
convolution
% then required is done with FFTs.
% 1988: L. Arevalo; 11.06.91 K. Schwarz, LNT
```

Erlangen; 6/12/91 csb.

```
%
N    = length(x);                %Sequence length
L    = 2^ceil(log((2*N-1))/log(2)); %FFT length
n    = 0:N-1;
W    = exp(-j*pi*n.*n/N);        %Chirp signal
FW   = fft([conj(W), zeros(1,L-2*N+1),
conj(W(N:-1:2))],L);
y    = ifft(FW.*fft(x.'.*W,L));  %Convolve using
FFT
y    = y(1:N).*W;                %Demodulate
```

Goertzel's Algorithm (or A Better DFT Algorithm)

Goertzel's algorithm [\[link\]](#), [\[link\]](#), [\[link\]](#) is another methods that calculates the DFT by converting it into a digital filtering problem. The method looks at the calculation of the DFT as the evaluation of a polynomial on the unit circle in the complex plane. This evaluation is done by Horner's method which is implemented recursively by an IIR filter.

The First-Order Goertzel Algorithm

The polynomial whose values on the unit circle are the DFT is a slightly modified z-transform of $x(n)$ given by

Equation:

$$X(z) = \sum_{n=0}^{N-1} x(n) z^{-n}$$

which for clarity in this development uses a positive exponent . This is illustrated for a length-4 sequence as a third-order polynomial by

Equation:

$$X(z) = x(3)z^3 + x(2)z^2 + x(1)z + x(0)$$

The DFT is found by evaluating [\[link\]](#) at $z = W^k$, which can be written as
Equation:

$$C(k) = X(z)|_{z=W^k} = DFT\{x(n)\}$$

where

Equation:

$$W = e^{-j2\pi/N}$$

The most efficient way of evaluating a general polynomial without any pre-processing is by “Horner’s rule” [\[link\]](#) which is a nested evaluation. This is illustrated for the polynomial in [\[link\]](#) by

Equation:

$$X(z) = \{[x(3)z + x(2)]z + x(1)\}z + x(0)$$

This nested sequence of operations can be written as a linear difference equation in the form of

Equation:

$$y(m) = zy(m-1) + x(N-m)$$

with initial condition $y(0) = 0$, and the desired result being the solution at $m = N$. The value of the polynomial is given by

Equation:

$$X(z) = y(N).$$

[\[link\]](#) can be viewed as a first-order IIR filter with the input being the data sequence in reverse order and the value of the polynomial at z being the

filter output sampled at $m = N$. Applying this to the DFT gives the Goertzel algorithm [\[link\]](#), [\[link\]](#) which is

Equation:

$$y(m) = W^k y(m-1) + x(N-m)$$

with $y(0) = 0$ and

Equation:

$$C(k) = y(N)$$

where

Equation:

$$C(k) = \sum_{n=0}^{N-1} x(n) W^{nk}.$$

The flowgraph of the algorithm can be found in [\[link\]](#), [\[link\]](#) and a simple FORTRAN program is given in the appendix.

When comparing this program with the direct calculation of [\[link\]](#), it is seen that the number of floating-point multiplications and additions are the same. In fact, the structures of the two algorithms look similar, but close examination shows that the way the sines and cosines enter the calculations is different. In [\[link\]](#), new sine and cosine values are calculated for each frequency and for each data value, while for the Goertzel algorithm in [\[link\]](#), they are calculated only for each frequency in the outer loop. Because of the recursive or feedback nature of the algorithm, the sine and cosine values are “updated” each loop rather than recalculated. This results in $2N$ trigonometric evaluations rather than $2N^2$. It also results in an increase in accumulated quantization error.

It is possible to modify this algorithm to allow entering the data in forward order rather than reverse order. The difference [\[link\]](#) becomes

Equation:

$$y(m) = z^{-1}y(m-1) + x(m-1)$$

if [\[link\]](#) becomes

Equation:

$$C(k) = z^{N-1}y(N)$$

for $y(0) = 0$. This is the algorithm programmed later.

The Second-Order Goertzel Algorithm

One of the reasons the first-order Goertzel algorithm does not improve efficiency is that the constant in the feedback or recursive path is complex and, therefore, requires four real multiplications and two real additions. A modification of the scheme to make it second-order removes the complex multiplications and reduces the number of required multiplications by two.

Define the variable $q(m)$ so that

Equation:

$$y(m) = q(m) - z^{-1}q(m-1).$$

This substituted into the right-hand side of [\[link\]](#) gives

Equation:

$$y(m) = zq(m-1) - q(m-2) + x(N-m).$$

Combining [\[link\]](#) and [\[link\]](#) gives the second order difference equation

Equation:

$$q(m) = (z + z^{-1})q(m-1) - q(m-2) + x(N-m)$$

which together with the output [\[link\]](#), comprise the second-order Goertzel algorithm where

Equation:

$$X(z) = y(N)$$

for initial conditions $q(0) = q(-1) = 0$.

A similar development starting with [\[link\]](#) gives a second-order algorithm with forward ordered input as

Equation:

$$q(m) = (z + z^{-1}) q(m-1) - q(m-2) + x(m-1)$$

Equation:

$$y(m) = q(m) - z q(-1)$$

with

Equation:

$$X(z) = z^{N-1} y(N)$$

and for $q(0) = q(-1) = 0$.

Note that both difference [\[link\]](#) and [\[link\]](#) are not changed if z is replaced with z^{-1} , only the output [\[link\]](#) and [\[link\]](#) are different. This means that the polynomial $X(z)$ may be evaluated at a particular z and its inverse z^{-1} from one solution of the difference [\[link\]](#) or [\[link\]](#) using the output equations

Equation:

$$X(z) = q(N) - z^{-1} q(N-1)$$

and

Equation:

$$X(1/z) = z^{N-1} (q(N) - z q(N-1)).$$

Clearly, this allows the DFT of a sequence to be calculated with half the arithmetic since the outputs are calculated two at a time. The second-order DE actually produces a solution $q(m)$ that contains two first-order components. The output equations are, in effect, zeros that cancel one or the other pole of the second-order solution to give the desired first-order solution. In addition to allowing the calculating of two outputs at a time, the second-order DE requires half the number of real multiplications as the first-order form. This is because the coefficient of the $q(m-2)$ is unity and the coefficient of the $q(m-1)$ is real if z and z^{-1} are complex conjugates of each other which is true for the DFT.

Analysis of Arithmetic Complexity and Timings

Analysis of the various forms of the Goertzel algorithm from their programs gives the following operation count for real multiplications and real additions assuming real data.

Algorithm	Real Mults.	Real Adds	Trig Eval.
Direct DFT	$4 N^2$	$4 N^2$	$2 N^2$
First-Order	$4 N^2$	$4 N^2 - 2N$	$2 N$
Second-Order	$2 N^2 + 2N$	$4 N^2$	$2 N$
Second-Order 2	$N^2 + N$	$2 N^2 + N$	N

Timings of the algorithms on a PC in milliseconds are given in the following table.

Algorithm	$N = 125$	$N = 257$
Direct DFT	4.90	19.83
First-Order	4.01	16.70
Second-Order	2.64	11.04
Second-Order 2	1.32	5.55

These timings track the floating point operation counts fairly well.

Conclusions

Goertzel's algorithm in its first-order form is not particularly interesting, but the two-at-a-time second-order form is significantly faster than a direct DFT. It can also be used for any polynomial evaluation or for the DTFT at unequally spaced values or for evaluating a few DFT terms. A very interesting observation is that the inner-most loop of the Glassman-Ferguson FFT [\[link\]](#) is a first-order Goertzel algorithm even though that FFT is developed in a very different framework.

In addition to floating-point arithmetic counts, the number of trigonometric function evaluations that must be made or the size of a table to store precomputed values should be considered. Since the value of the W^{nk}

terms in [\[link\]](#) are iteratively calculate in the IIR filter structure, there is round-off error accumulation that should be analyzed in any application.

It may be possible to further improve the efficiency of the second-order Goertzel algorithm for calculating all of the DFT of a number sequence. Perhaps a fourth order DE could calculate four output values at a time and they could be separated by a numerator that would cancel three of the zeros. Perhaps the algorithm could be arranged in stages to give an $N \log(N)$ operation count. The current algorithm does not take into account any of the symmetries of the input index. Perhaps some of the ideas used in developing the QFT [\[link\]](#), [\[link\]](#), [\[link\]](#) could be used here.

The Quick Fourier Transform (QFT)

One stage of the QFT can use the symmetries of the sines and cosines to calculate a DFT more efficiently than directly implementing the definition [Multidimensional Index Mapping: Equation 1](#). Similar to the Goertzel algorithm, the one-stage QFT is a better N^2 DFT algorithm for arbitrary lengths. See [The Cooley-Tukey Fast Fourier Transform Algorithm: The Quick Fourier Transform, An FFT based on Symmetries](#).

Factoring the Signal Processing Operators

A third approach to removing redundancy in an algorithm is to express the algorithm as an operator and then factor that operator into sparse factors. This approach is used by Tolimieri [\[link\]](#), [\[link\]](#), Egner [\[link\]](#), Selesnick, Elliott [\[link\]](#) and others. It is presented in a more general form in [DFT and FFT: An Algebraic View](#). The operators may be in the form of a matrix or a tensor operator.

The FFT from Factoring the DFT Operator

The definition of the DFT in [Multidimensional Index Mapping: Equation 1](#) can be written as a matrix-vector operation by $C = WX$ which, for $N = 8$ is

Equation:

$$\begin{array}{rcl}
 C(0) & & W^0 \ W^0 \ W^0 \ W^0 \ W^0 \ W^0 \ W^0 \ W^0 \ x(0) \\
 C(1) & & W^0 \ W^1 \ W^2 \ W^3 \ W^4 \ W^5 \ W^6 \ W^7 \ x(1) \\
 C(2) & & W^0 \ W^2 \ W^4 \ W^6 \ W^8 \ W^{10} \ W^{12} \ W^{14} \ x(2) \\
 C(3) & & W^0 \ W^3 \ W^6 \ W^9 \ W^{12} \ W^{15} \ W^{18} \ W^{21} \ x(3) \\
 C(4) & = & W^0 \ W^4 \ W^8 \ W^{12} \ W^{16} \ W^{20} \ W^{24} \ W^{28} \ x(4) \\
 C(5) & & W^0 \ W^5 \ W^{10} \ W^{15} \ W^{20} \ W^{25} \ W^{30} \ W^{35} \ x(5) \\
 C(6) & & W^0 \ W^6 \ W^{12} \ W^{18} \ W^{24} \ W^{30} \ W^{36} \ W^{42} \ x(6) \\
 C(7) & & W^0 \ W^7 \ W^{14} \ W^{21} \ W^{28} \ W^{35} \ W^{42} \ W^{49} \ x(7)
 \end{array}$$

which clearly requires $N^2 = 64$ complex multiplications and $N(N - 1)$ additions. A factorization of the DFT operator, W , gives $W = F_1 \ F_2 \ F_3$ and $C = F_1 \ F_2 \ F_3 \ X$ or, expanded,

Equation:

$$\begin{array}{rcl}
 C(0) & & 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 C(4) & & 1 \ -1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \\
 C(2) & & 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ W^0 \ 0 \ -W^2 \ 0 \ 0 \ 0 \ 0 \ 0 \\
 C(6) & = & 0 \ 0 \ 1 \ -1 \ 0 \ 0 \ 0 \ 0 \ 0 \ W^0 \ 0 \ -W^2 \ 0 \ 0 \ 0 \ 0 \\
 C(1) & & 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \\
 C(5) & & 0 \ 0 \ 0 \ 0 \ 1 \ -1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \\
 C(3) & & 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ W^0 \ 0 \ -W^0 \ 0 \\
 C(7) & & 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ -1 \ 0 \ 0 \ 0 \ 0 \ 0 \ W^2 \ 0 \ -W^2
 \end{array}$$

Equation:

$$\begin{array}{cccccccc}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & x(0) \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & x(1) \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & x(2) \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & x(3) \\
W^0 & 0 & 0 & 0 & -W^0 & 0 & 0 & 0 & x(4) \\
0 & W^1 & 0 & 0 & 0 & -W^1 & 0 & 0 & x(5) \\
0 & 0 & W^2 & 0 & 0 & 0 & -W^2 & 0 & x(6) \\
0 & 0 & 0 & W^3 & 0 & 0 & 0 & -W^3 & x(7)
\end{array}$$

where the F_i matrices are sparse. Note that each has 16 (or $2N$) non-zero terms and F_2 and F_3 have 8 (or N) non-unity terms. If $N = 2^M$, then the number of factors is $\log(N) = M$. In another form with the twiddle factors separated so as to count the complex multiplications we have

Equation:

$$\begin{array}{rcl}
C(0) & & 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
C(4) & & 1 \ -1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\
C(2) & & 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \\
C(6) & = & 0 \ 0 \ 1 \ -1 \ 0 \ 0 \ 0 \ 0 \\
C(1) & & 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \\
C(5) & & 0 \ 0 \ 0 \ 0 \ 1 \ -1 \ 0 \ 0 \\
C(3) & & 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \\
C(7) & & 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ -1
\end{array}$$

Equation:

$$\begin{array}{cccccccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & W^0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & W^2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & W^0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & W^2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1
\end{array}$$

Equation:

$$\begin{array}{cccccccccccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & x(0) \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & x(1) \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & x(2) \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & x(3) \\
0 & 0 & 0 & 0 & W^0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & x(4) \\
0 & 0 & 0 & 0 & 0 & W^1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & x(5) \\
0 & 0 & 0 & 0 & 0 & 0 & W^2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & x(6) \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & W^3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & x(7)
\end{array}$$

which is in the form $C = A_1 M_1 A_2 M_2 A_3 X$ described by the index map. A_1 , A_2 , and A_3 each represents 8 additions, or, in general, N additions. M_1 and M_2 each represent 4 (or $N/2$) multiplications.

This is a very interesting result showing that implementing the DFT using the factored form requires considerably less arithmetic than the single factor definition. Indeed, the form of the formula that Cooley and Tukey derived showing that the amount of arithmetic required by the FFT is on the order of $N \log(N)$ can be seen from the factored operator formulation.

Much of the theory of the FFT can be developed using operator factoring and it has some advantages for implementation of parallel and vector computer architectures. The eigenspace approach is somewhat of the same type [\[link\]](#).

Algebraic Theory of Signal Processing Algorithms

A very general structure for all kinds of algorithms can be generalized from the approach of operators and operator decomposition. This is developed as "Algebraic Theory of Signal Processing" discussed in the module [DFT and FFT: An Algebraic View](#) by Püschel and others [\[link\]](#).

Winograd's Short DFT Algorithms

In 1976, S. Winograd [\[link\]](#) presented a new DFT algorithm which had significantly fewer multiplications than the Cooley-Tukey FFT which had been published eleven years earlier. This new Winograd Fourier Transform Algorithm (WFTA) is based on the type- one index map from [Multidimensional Index Mapping](#) with each of the relatively prime length short DFT's calculated by very efficient special algorithms. It is these short algorithms that this section will develop. They use the index permutation of Rader described in the another module to convert the prime length short DFT's into cyclic convolutions. Winograd developed a method for calculating digital convolution with the minimum number of multiplications. These optimal algorithms are based on the polynomial residue reduction techniques of [Polynomial Description of Signals: Equation 1](#) to break the convolution into multiple small ones [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#).

The operation of discrete convolution defined by
Equation:

$$y(n) = \sum_k h(n - k) x(k)$$

is called a **bilinear** operation because, for a fixed $h(n)$, $y(n)$ is a linear function of $x(n)$ and for a fixed $x(n)$ it is a linear function of $h(n)$. The operation of cyclic convolution is the same but with all indices evaluated modulo N .

Recall from [Polynomial Description of Signals: Equation 3](#) that length- N cyclic convolution of $x(n)$ and $h(n)$ can be represented by polynomial multiplication

Equation:

$$Y(s) = X(s) H(s) \mod (s^N - 1)$$

This bilinear operation of [\[link\]](#) and [\[link\]](#) can also be expressed in terms of linear matrix operators and a simpler bilinear operator denoted by o which may be only a simple element-by-element multiplication of the two vectors [\[link\]](#), [\[link\]](#), [\[link\]](#). This matrix formulation is

Equation:

$$Y = C[AXoBH]$$

where X , H and Y are length- N vectors with elements of $x(n)$, $h(n)$ and $y(n)$ respectively. The matrices A and B have dimension $M \times N$, and C is $N \times M$ with $M \geq N$. The elements of A , B , and C are constrained to be simple; typically small integers or rational numbers. It will be these matrix operators that do the equivalent of the residue reduction on the polynomials in [\[link\]](#).

In order to derive a useful algorithm of the form [\[link\]](#) to calculate [\[link\]](#), consider the polynomial formulation [\[link\]](#) again. To use the residue reduction scheme, the modulus is factored into relatively prime factors. Fortunately the factoring of this particular polynomial, $s^N - 1$, has been extensively studied and it has considerable structure. When factored over the rationals, which means that the only coefficients allowed are rational numbers, the factors are called cyclotomic polynomials [\[link\]](#), [\[link\]](#), [\[link\]](#). The most interesting property for our purposes is that most of the coefficients of cyclotomic polynomials are zero and the others are plus or minus unity for degrees up to over one hundred. This means the residue reduction will generally require no multiplications.

The operations of reducing $X(s)$ and $H(s)$ in [\[link\]](#) are carried out by the matrices A and B in [\[link\]](#). The convolution of the residue polynomials is carried out by the o operator and the recombination by the CRT is done by the C matrix. More details are in [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#) but the important fact is the A and B matrices usually contain only zero and plus or minus unity entries and the C matrix only contains rational numbers. The only general multiplications are those represented by o . Indeed, in the theoretical results from computational complexity theory, these real or complex multiplications are usually the only ones counted. In practical

algorithms, the rational multiplications represented by C could be a limiting factor.

The $h(n)$ terms are fixed for a digital filter, or they represent the W terms from [Multidimensional Index Mapping: Equation 1](#) if the convolution is being used to calculate a DFT. Because of this, $d = BH$ in [\[link\]](#) can be precalculated and only the A and C operators represent the mathematics done at execution of the algorithm. In order to exploit this feature, it was shown [\[link\]](#), [\[link\]](#) that the properties of [\[link\]](#) allow the exchange of the more complicated operator C with the simpler operator B . Specifically this is given by

Equation:

$$Y = C[AXoBH]$$

Equation:

$$Y' = B^T [AXoC^T H']$$

where H' has the same elements as H , but in a permuted order, and likewise Y' and Y . This very important property allows precomputing the more complicated $C^T H'$ in [\[link\]](#) rather than BH as in [\[link\]](#).

Because BH or $C^T H'$ can be precomputed, the bilinear form of [\[link\]](#) and [\[link\]](#) can be written as a linear form. If an $M \times M$ diagonal matrix D is formed from $d = C^T H'$, or in the case of [\[link\]](#), $d = BH$, assuming a commutative property for o , [\[link\]](#) becomes

Equation:

$$Y' = B^T DAX$$

and [\[link\]](#) becomes

Equation:

$$Y = CDAX$$

In most cases there is no reason not to use the same reduction operations on X and H , therefore, B can be the same as A and [\[link\]](#) then becomes

Equation:

$$Y' = A^T D A X$$

In order to illustrate how the residue reduction is carried out and how the A matrix is obtained, the length-5 DFT algorithm started in [The DFT as Convolution or Filtering: Matrix 1](#) will be continued. The DFT is first converted to a length-4 cyclic convolution by the index permutation from [The DFT as Convolution or Filtering: Equation 3](#) to give the cyclic convolution in [The DFT as Convolution or Filtering](#). To avoid confusion from the permuted order of the data $x(n)$ in [The DFT as Convolution or Filtering](#), the cyclic convolution will first be developed without the permutation, using the polynomial $U(s)$

Equation:

$$U(s) = x(1) + x(3)s + x(4)s^2 + x(2)s^3$$

Equation:

$$U(s) = u(0) + u(1)s + u(2)s^2 + u(3)s^3$$

and then the results will be converted back to the permuted $x(n)$. The length-4 cyclic convolution in terms of polynomials is

Equation:

$$Y(s) = U(s) H(s) \mod (s^4 - 1)$$

and the modulus factors into three cyclotomic polynomials

Equation:

$$s^4 - 1 = (s^2 - 1)(s^2 + 1)$$

Equation:

$$= (s - 1) (s + 1) (s^2 + 1)$$

Equation:

$$= P_1 P_2 P_3$$

Both $U(s)$ and $H(s)$ are reduced modulo these three polynomials. The reduction modulo P_1 and P_2 is done in two stages. First it is done modulo $(s^2 - 1)$, then that residue is further reduced modulo $(s - 1)$ and $(s + 1)$.

Equation:

$$U(s) = u_0 + u_1 s + u_2 s^2 + u_3 s^3$$

Equation:

$$U'(s) = ((U(s)))_{(s^2-1)} = (u_0 + u_2) + (u_1 + u_3)s$$

Equation:

$$U1(s) = ((U'(s)))_{P_1} = (u_0 + u_1 + u_2 + u_3)$$

Equation:

$$U2(s) = ((U'(s)))_{P_2} = (u_0 - u_1 + u_2 - u_3)$$

Equation:

$$U3(s) = ((U(s)))_{P_3} = (u_0 - u_2) + (u_1 - u_3)s$$

The reduction in [\[link\]](#) of the data polynomial [\[link\]](#) can be denoted by a matrix operation on a vector which has the data as entries.

Equation:

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} u_0 + u_2 \\ u_1 + u_3 \end{bmatrix}$$

and the reduction in [\[link\]](#) is

Equation:

$$\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} u_0 - u_2 \\ u_1 - u_3 \end{bmatrix}$$

Combining [\[link\]](#) and [\[link\]](#) gives one operator

Equation:

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} u_0 + u_2 \\ u_1 + u_3 \\ u_0 - u_2 \\ u_1 - u_3 \end{bmatrix} = \begin{bmatrix} u_0 + u_2 \\ u_1 + u_3 \\ u_0 - u_2 \\ u_1 - u_3 \end{bmatrix} = \begin{bmatrix} w_0 \\ w_1 \\ v_0 \\ v_1 \end{bmatrix}$$

Further reduction of $v_0 + v_1 s$ is not possible because $P_3 = s^2 + 1$ cannot be factored over the rationals. However $s^2 - 1$ can be factored into $P_1 P_2 = (s - 1)(s + 1)$ and, therefore, $w_0 + w_1 s$ can be further reduced as was done in [\[link\]](#) and [\[link\]](#) by

Equation:

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = w_0 + w_1 = u_0 + u_2 + u_1 + u_3$$

Equation:

$$\begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = w_0 - w_1 = u_0 + u_2 - u_1 - u_3$$

Combining [\[link\]](#), [\[link\]](#) and [\[link\]](#) gives

Equation:

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ v_0 \\ v_1 \end{bmatrix}$$

The same reduction is done to $H(s)$ and then the convolution of [\[link\]](#) is done by multiplying each residue polynomial of $X(s)$ and $H(s)$ modulo each corresponding cyclotomic factor of $P(s)$ and finally a recombination using the polynomial Chinese Remainder Theorem (CRT) as in [Polynomial Description of Signals: Equation 9](#) and [Polynomial Description of Signals: Equation 13](#).

Equation:

$$Y(s) = K_1(s)U_1(s)H_1(s) + K_2(s)U_2(s)H_2(s) + K_3(s)U_3(s)H_3(s)$$

$$\text{mod } (s^4 - 1)$$

where $U_1(s) = r_1$ and $U_2(s) = r_2$ are constants and $U_3(s) = v_0 + v_1s$ is a first degree polynomial. U_1 times H_1 and U_2 times H_2 are easy, but multiplying U_3 time H_3 modulo $(s^2 + 1)$ is more difficult.

The multiplication of $U_3(s)$ times $H_3(s)$ can be done by the Toom-Cook algorithm [\[link\]](#), [\[link\]](#), [\[link\]](#) which can be viewed as Lagrange interpolation or polynomial multiplication modulo a special polynomial with three arbitrary coefficients. To simplify the arithmetic, the constants are chosen to be plus and minus one and zero. The details of this can be found in [\[link\]](#), [\[link\]](#), [\[link\]](#). For this example it can be verified that

Equation:

$$((v_0 + v_1 s)(h_0 + h_1 s))_{s^2+1} = (v_0 h_0 - v_1 h_1) + (v_0 h_1 + v_1 h_0)s$$

which by the Toom-Cook algorithm or inspection is

Equation:

$$\begin{bmatrix} 1 & -1 & 0 \\ -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \end{bmatrix} o \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}$$

where o signifies point-by-point multiplication. The total A matrix in [\[link\]](#) is a combination of [\[link\]](#) and [\[link\]](#) giving

Equation:

$$AX = A_1 A_2 A_3 X$$

Equation:

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ v_0 \\ v_1 \end{bmatrix}$$

where the matrix A_3 gives the residue reduction $s^2 - 1$ and $s^2 + 1$, the upper left-hand part of A_2 gives the reduction modulo $s - 1$ and $s + 1$, and the lower right-hand part of A_1 carries out the Toom-Cook algorithm modulo $s^2 + 1$ with the multiplication in [\[link\]](#). Notice that by calculating [\[link\]](#) in the three stages, seven additions are required. Also notice that A_1 is not square. It is this “expansion” that causes more than N multiplications to be required in o in [\[link\]](#) or D in [\[link\]](#). This staged reduction will derive the A operator for [\[link\]](#)

The method described above is very straight-forward for the shorter DFT lengths. For $N = 3$, both of the residue polynomials are constants and the

multiplication given by 0 in [\[link\]](#) is trivial. For $N = 5$, which is the example used here, there is one first degree polynomial multiplication required but the Toom-Cook algorithm uses simple constants and, therefore, works well as indicated in [\[link\]](#). For $N = 7$, there are two first degree residue polynomials which can each be multiplied by the same techniques used in the $N = 5$ example. Unfortunately, for any longer lengths, the residue polynomials have an order of three or greater which causes the Toom-Cook algorithm to require constants of plus and minus two and worse. For that reason, the Toom-Cook method is not used, and other techniques such as index mapping are used that require more than the minimum number of multiplications, but do not require an excessive number of additions. The resulting algorithms still have the structure of [\[link\]](#). Blahut [\[link\]](#) and Nussbaumer [\[link\]](#) have a good collection of algorithms for polynomial multiplication that can be used with the techniques discussed here to construct a wide variety of DFT algorithms.

The constants in the diagonal matrix D can be found from the CRT matrix C in [\[link\]](#) using $d = C^T H'$ for the diagonal terms in D . As mentioned above, for the smaller prime lengths of 3, 5, and 7 this works well but for longer lengths the CRT becomes very complicated. An alternate method for finding D uses the fact that since the linear form [\[link\]](#) or [\[link\]](#) calculates the DFT, it is possible to calculate a known DFT of a given $x(n)$ from the definition of the DFT in [Multidimensional Index Mapping: Equation 1](#) and, given the A matrix in [\[link\]](#), solve for D by solving a set of simultaneous equations. The details of this procedure are described in [\[link\]](#).

A modification of this approach also works for a length which is an odd prime raised to some power: $N = P^M$. This is a bit more complicated [\[link\]](#), [\[link\]](#) but has been done for lengths of 9 and 25. For longer lengths, the conventional Cooley-Tukey type- two index map algorithm seems to be more efficient. For powers of two, there is no primitive root, and therefore, no simple conversion of the DFT into convolution. It is possible to use two generators [\[link\]](#), [\[link\]](#), [\[link\]](#) to make the conversion and there exists a set of length 4, 8, and 16 DFT algorithms of the form in [\[link\]](#) in [\[link\]](#).

In [\[link\]](#) an operation count of several short DFT algorithms is presented. These are practical algorithms that can be used alone or in conjunction with the index mapping to give longer DFT's as shown in [The Prime Factor and](#)

[Winograd Fourier Transform Algorithms](#). Most are optimized in having either the theoretical minimum number of multiplications or the minimum number of multiplications without requiring a very large number of additions. Some allow other reasonable trade-offs between numbers of multiplications and additions. There are two lists of the number of multiplications. The first is the number of actual floating point multiplications that must be done for that length DFT. Some of these (one or two in most cases) will be by rational constants and the others will be by irrational constants. The second list is the total number of multiplications given in the diagonal matrix D in [\[link\]](#). At least one of these will be unity (the one associated with $X(0)$) and in some cases several will be unity (for $N = 2^M$). The second list is important in programming the WFTA in [The Prime Factor and Winograd Fourier Transform Algorithm: The Winograd Fourier Transform Algorithm](#).

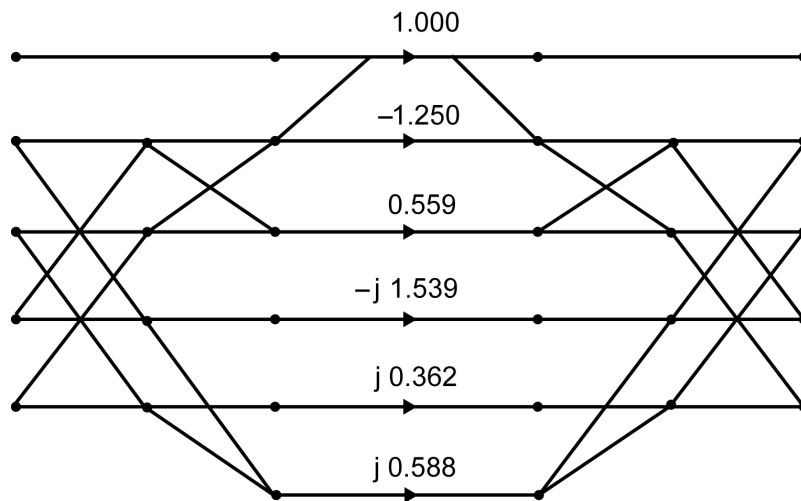
Length N	Mult Non-one	Mult Total	Adds
2	0	4	4
3	4	6	12
4	0	8	16
5	10	12	34
7	16	18	72
8	4	16	52
9	20	22	84
11	40	42	168

13	40	42	188
16	20	36	148
17	70	72	314
19	76	78	372
25	132	134	420
32	68	-	388

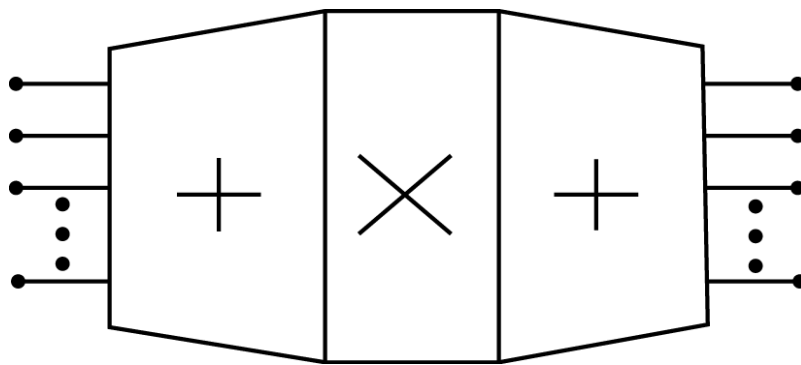
Number of Real Multiplications and Additions for a Length- N DFT of Complex Data

Because of the structure of the short DFTs, the number of real multiplications required for the DFT of real data is exactly half that required for complex data. The number of real additions required is slightly less than half that required for complex data because $(N - 1)$ of the additions needed when N is prime add a real to an imaginary, and that is not actually performed. When $N = 2m$, there are $(N - 2)$ of these pseudo additions. The special case for real data is discussed in [\[link\]](#), [\[link\]](#), [\[link\]](#).

The structure of these algorithms are in the form of $X' = CDAX$ or $B^T DAX$ or $A^T DAX$ from [\[link\]](#) and [\[link\]](#). The A and B matrices are generally M by N with $M \geq N$ and have elements that are integers, generally 0 or ± 1 . A pictorial description is given in [\[link\]](#).



Flow Graph for the Length-5 DFT



Block Diagram of a Winograd Short DFT

The flow graph in [\[link\]](#) should be compared with the matrix description of [\[link\]](#) and [\[link\]](#), and with the programs in [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#) and the appendices. The shape in [\[link\]](#) illustrates the expansion of the data by A . That is to say, AX has more entries than X because $M > N$. The A operator consists of additions, the D operator gives the M multiplications (some by one) and A^T contracts the data back to N values with additions only. M is one half the second list of multiplies in [\[link\]](#).

An important characteristic of the D operator in the calculation of the DFT is its entries are either purely real or imaginary. The reduction of the W vector by $(s^{(N-1)/2} - 1)$ and $(s^{(N-1)/2} + 1)$ separates the real and the imaginary constants. This is discussed in [\[link\]](#), [\[link\]](#). The number of multiplications for complex data is only twice those necessary for real data, not four times.

Although this discussion has been on the calculation of the DFT, very similar results are true for the calculation of convolution and correlation, and these will be further developed in [Algorithms for Data with Restrictions](#). The $A^T D A$ structure and the picture in [\[link\]](#) are the same for convolution. Algorithms and operation counts can be found in [\[link\]](#), [\[link\]](#), [\[link\]](#).

The Bilinear Structure

The bilinear form introduced in [\[link\]](#) and the related linear form in [\[link\]](#) are very powerful descriptions of both the DFT and convolution.

Equation:

$$\text{Bilinear: } Y = C[AX \circ BH]$$

Equation:

$$\text{Linear: } Y = CDA X$$

Since [\[link\]](#) is a bilinear operation defined in terms of a second bilinear operator \circ , this formulation can be nested. For example if \circ is itself defined in terms of a second bilinear operator $@$, by

Equation:

$$X \circ H = C' [A' X @ B' H]$$

then [\[link\]](#) becomes

Equation:

$$Y = CC' [A'AX @ B'BH]$$

For convolution, if A represents the polynomial residue reduction modulo the cyclotomic polynomials, then A is square (e.g. [\[link\]](#)) and \circ represents multiplication of the residue polynomials modulo the cyclotomic polynomials. If A represents the reduction modulo the cyclotomic polynomials plus the Toom-Cook reduction as was the case in the example of [\[link\]](#), then A is $N \times M$ and \circ is term-by-term simple scalar multiplication. In this case AX can be thought of as a transform of X and C is the inverse transform. This is called a rectangular transform [\[link\]](#) because A is rectangular. The transform requires only additions and convolution is done with M multiplications. The other extreme is when A represents reduction over the N complex roots of $s^N - 1$. In this case A is the DFT itself, as in the example of (43), and \circ is point by point complex multiplication and C is the inverse DFT. A trivial case is where A , B and C are identity operators and \circ is the cyclic convolution.

This very general and flexible bilinear formulation coupled with the idea of nesting in [\[link\]](#) gives a description of most forms of convolution.

Winograd's Complexity Theorems

Because Winograd's work [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#) has been the foundation of the modern results in efficient convolution and DFT algorithms, it is worthwhile to look at his theoretical conclusions on optimal algorithms. Most of his results are stated in terms of polynomial multiplication as [Polynomial Description of Signals: Equation 3](#) or [\[link\]](#). The measure of computational complexity is usually the number of multiplications, and only certain multiplications are counted. This must be understood in order not to misinterpret the results.

This section will simply give a statement of the pertinent results and will not attempt to derive or prove anything. A short interpretation of each theorem will be given to relate the result to the algorithms developed in this chapter. The indicated references should be consulted for background and detail.

Theorem 1 [\[link\]](#) Given two polynomials, $x(s)$ and $h(s)$, of degree N and M respectively, each with indeterminate coefficients that are elements of a field H , $N + M + 1$ multiplications are necessary to compute the coefficients of the product polynomial $x(s)h(s)$. Multiplication by elements of the field G (the field of constants), which is contained in H , are not counted and G contains at least $N + M$ distinct elements.

The upper bound in this theorem can be realized by choosing an arbitrary modulus polynomial $P(s)$ of degree $N + M + 1$ composed of $N + M + 1$ distinct linear polynomial factors with coefficients in G which, since its degree is greater than the product $x(s)h(s)$, has no effect on the product, and by reducing $x(s)$ and $h(s)$ to $N + M + 1$ residues modulo the $N + M + 1$ factors of $P(s)$. These residues are multiplied by each other, requiring $N + M + 1$ multiplications, and the results recombined using the Chinese remainder theorem (CRT). The operations required in the reduction and recombination are not counted, while the residue multiplications are. Since the modulus $P(s)$ is arbitrary, its factors are chosen to be simple so as to make the reduction and CRT simple. Factors of zero, plus and minus unity, and infinity are the simplest. Plus and minus two and other factors complicate the actual calculations considerably, but the theorem does not take that into account. This algorithm is a form of the Toom-Cook algorithm and of Lagrange interpolation [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#). For our applications, H is the field of reals and G the field of rationals.

Theorem 2 [\[link\]](#) If an algorithm exists which computes $x(s)h(s)$ in $N + M + 1$ multiplications, all but one of its multiplication steps must necessarily be of the form

Equation:

$$mk = (gk' + x(gk)) (gk'' + h(gk)) \quad \text{for } k = 0, 1, \dots, N + M$$

where g_k are distinct elements of G ; and g_k' and g_k'' are arbitrary elements of G

This theorem states that the structure of an optimal algorithm is essentially unique although the factors of $P(s)$ may be chosen arbitrarily.

Theorem 3 [\[link\]](#) Let $P(s)$ be a polynomial of degree N and be of the form $P(s) = Q(s)k$, where $Q(s)$ is an irreducible polynomial with coefficients in G and k is a positive integer. Let $x(s)$ and $h(s)$ be two polynomials of degree at least $N - 1$ with coefficients from H , then $2N - 1$ multiplications are required to compute the product $x(s)h(s)$ modulo $P(s)$.

This theorem is similar to [Theorem 1](#) with the operations of the reduction of the product modulo $P(s)$ not being counted.

Theorem 4 [\[link\]](#) Any algorithm that computes the product $x(s)h(s)$ modulo $P(s)$ according to the conditions stated in Theorem 3 and requires $2N - 1$ multiplications will necessarily be of one of three structures, each of which has the form of Theorem 2 internally.

As in [Theorem 2](#), this theorem states that only a limited number of possible structures exist for optimal algorithms.

Theorem 5 [\[link\]](#) If the modulus polynomial $P(s)$ has degree N and is not irreducible, it can be written in a unique factored form $P(s) = P_1^{m_1}(s)P_2^{m_2}(s)\dots P_k^{m_k}(s)$ where each of the $P_i(s)$ are irreducible over the allowed coefficient field G . $2N - k$ multiplications are necessary to compute the product $x(s)h(s)$ modulo $P(s)$ where $x(s)$ and $h(s)$ have coefficients in H and are of degree at least $N - 1$. All algorithms that calculate this product in $2N - k$ multiplications must be of a form where each of the k residue polynomials of $x(s)$ and $h(s)$ are separately multiplied modulo the factors of $P(s)$ via the CRT.

Corollary: If the modulus polynomial is $P(s) = s^N - 1$, then $2N - t(N)$ multiplications are necessary to compute $x(s)h(s)$ modulo $P(s)$, where $t(N)$ is the number of positive divisors of N .

[Theorem 5](#) is very general since it allows a general modulus polynomial. The proof of the upper bound involves reducing $x(s)$ and $h(s)$ modulo the k factors of $P(s)$. Each of the k irreducible residue polynomials is then multiplied using the method of [Theorem 4](#) requiring $2Ni - 1$ multiplies and the products are combined using the CRT. The total number of multiplies from the k parts is $2N - k$. The theorem also states the structure of these

optimal algorithms is essentially unique. The special case of $P(s) = s^N - 1$ is interesting since it corresponds to cyclic convolution and, as stated in the corollary, k is easily determined. The factors of $s^N - 1$ are called cyclotomic polynomials and have interesting properties [\[link\]](#), [\[link\]](#), [\[link\]](#).

Theorem 6 [\[link\]](#), [\[link\]](#) Consider calculating the DFT of a prime length real-valued number sequence. If G is chosen as the field of rational numbers, the number of real multiplications necessary to calculate a length- P DFT is $u(DFT(N)) = 2P - 3 - t(P - 1)$ where $t(P - 1)$ is the number of divisors of $P - 1$.

This theorem not only gives a lower limit on any practical prime length DFT algorithm, it also gives practical algorithms for $N = 3, 5$, and 7 . Consider the operation counts in [\[link\]](#) to understand this theorem. In addition to the real multiplications counted by complexity theory, each optimal prime-length algorithm will have one multiplication by a rational constant. That constant corresponds to the residue modulo $(s-1)$ which always exists for the modulus $P(s) = s^{N-1} - 1$. In a practical algorithm, this multiplication must be carried out, and that accounts for the difference in the prediction of [Theorem 6](#) and count in [\[link\]](#). In addition, there is another operation that for certain applications must be counted as a multiplication. That is the calculation of the zero frequency term $X(0)$ in the first row of the example in [The DFT as Convolution or Filtering: Matrix 1](#). For applications to the WFTA discussed in [The Prime Factor and Winograd Fourier Transform Algorithms: The Winograd Fourier Transform Algorithm](#), that operation must be counted as a multiply. For lengths longer than 7 , optimal algorithms require too many additions, so compromise structures are used.

Theorem 7 [\[link\]](#), [\[link\]](#) If G is chosen as the field of rational numbers, the number of real multiplications necessary to calculate a length- N DFT where N is a prime number raised to an integer power: $N = P^m$, is given by
Equation:

$$u(DFT(N)) = 2N - ((m+1)/2)t(P - 1) - m - 1$$

where $t(P - 1)$ is the number of divisors of $(P - 1)$.

This result seems to be practically achievable only for $N = 9$, or perhaps 25. In the case of $N = 9$, there are two rational multiplies that must be carried out and are counted in [\[link\]](#) but are not predicted by [Theorem 7](#). Experience [\[link\]](#) indicates that even for $N = 25$, an algorithm based on a Cooley-Tukey FFT using a type 2 index map gives an over-all more balanced result.

Theorem 8 [\[link\]](#) If G is chosen as the field of rational numbers, the number of real multiplications necessary to calculate a length- N DFT where $N = 2m$ is given by

Equation:

$$u(DFT(N)) = 2N - m^2 - m - 2$$

This result is not practically useful because the number of additions necessary to realize this minimum of multiplications becomes very large for lengths greater than 16. Nevertheless, it proves the minimum number of multiplications required of an optimal algorithm is a linear function of N rather than of $N \log N$ which is that required of practical algorithms. The best practical power-of-two algorithm seems to be the Split-Radix [\[link\]](#) FFT discussed in [The Cooley-Tukey Fast Fourier Transform Algorithm: The Split-Radix FFT Algorithm](#).

All of these theorems use ideas based on residue reduction, multiplication of the residues, and then combination by the CRT. It is remarkable that this approach finds the minimum number of required multiplications by a constructive proof which generates an algorithm that achieves this minimum; and the structure of the optimal algorithm is, within certain variations, unique. For shorter lengths, the optimal algorithms give practical programs. For longer lengths the uncounted operations involved with the multiplication of the higher degree residue polynomials become very large and impractical. In those cases, efficient suboptimal algorithms can be generated by using the same residue reduction as for the optimal case, but by using methods other than the Toom-Cook algorithm of [Theorem 1](#) to multiply the residue polynomials.

Practical long DFT algorithms are produced by combining short prime length optimal DFT's with the Type 1 index map from [Multidimensional Index Mapping](#) to give the Prime Factor Algorithm (PFA) and the Winograd Fourier Transform Algorithm (WFTA) discussed in [The Prime Factor and Winograd Fourier Transform Algorithms](#). It is interesting to note that the index mapping technique is useful inside the short DFT algorithms to replace the Toom-Cook algorithm and outside to combine the short DFT's to calculate long DFT's.

The Automatic Generation of Winograd's Short DFTs

by Ivan Selesnick, Polytechnic Institute of New York University

Introduction

Efficient prime length DFTs are important for two reasons. A particular application may require a prime length DFT and secondly, the maximum length and the variety of lengths of a PFA or WFTA algorithm depend upon the availability of prime length modules.

This [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#) discusses automation of the process Winograd used for constructing prime length FFTs [\[link\]](#), [\[link\]](#) for $N < 7$ and that Johnson and Burrus [\[link\]](#) extended to $N < 19$. It also describes a program that will design any prime length FFT in principle, and will also automatically generate the algorithm as a C program and draw the corresponding flow graph.

Winograd's approach uses Rader's method to convert a prime length DFT into a $P - 1$ length cyclic convolution, polynomial residue reduction to decompose the problem into smaller convolutions [\[link\]](#), [\[link\]](#), and the Toom-Cook algorithm [\[link\]](#), [\[link\]](#). The Chinese Remainder Theorem (CRT) for polynomials is then used to recombine the shorter convolutions. Unfortunately, the design procedure derived directly from Winograd's theory becomes cumbersome for longer length DFTs, and this has often prevented the design of DFT programs for lengths greater than 19.

Here we use three methods to facilitate the construction of prime length FFT modules. First, the matrix exchange property [\[link\]](#), [\[link\]](#), [\[link\]](#) is used so that the transpose of the reduction operator can be used rather than the more complicated CRT reconstruction operator. This is then combined with the numerical method [\[link\]](#) for obtaining the multiplication coefficients rather than the direct use of the CRT. We also deviate from the Toom-Cook algorithm, because it requires too many additions for the lengths in which we are interested. Instead we use an iterated polynomial multiplication algorithm [\[link\]](#). We have incorporated these three ideas into a single structural procedure that automates the design of prime length FFTs.

Matrix Description

It is important that each step in the Winograd FFT can be described using matrices. By expressing cyclic convolution as a bilinear form, a compact form of prime length DFTs can be obtained.

If y is the cyclic convolution of h and x , then y can be expressed as
Equation:

$$y = C[Ax.*Bh]$$

where, using the Matlab convention, $.*$ represents point by point multiplication. When A, B , and C are allowed to be complex, A and B are seen to be the DFT operator and C , the inverse DFT. When only real numbers are allowed, A , B , and C will be rectangular. This form of convolution is presented with many examples in [\[link\]](#). Using the matrix exchange property explained in [\[link\]](#) and [\[link\]](#) this form can be written as
Equation:

$$y = RB^T [C^T Rh.*Ax]$$

where R is the permutation matrix that reverses order.

When h is fixed, as it is when considering prime length DFTs, the term $C^T R h$ can be precomputed and a diagonal matrix D formed by $D = \text{diag}\{C^T R h\}$. This is advantageous because in general, C is more complicated than B , so the ability to “hide” C saves computation. Now $y = R B^T D A x$ or $y = R A^T D A x$ since A and B can be the same; they implement a polynomial reduction. The form $y = R^T D A x$ can also be used for the prime length DFTs, it is only necessary to permute the entries of x and to ensure that the DC term is computed correctly. The computation of the DC term is simple, for the residue of a polynomial modulo $a - 1$ is always the sum of the coefficients. After adding the x_0 term of the original input sequence, to the $s - l$ residue, the DC term is obtained. Now $DFT\{x\} = R A^T D A x$. In [\[link\]](#) Johnson observes that by permuting the elements on the diagonal of D , the output can be permuted, so that the R matrix can be hidden in D , and $DFT\{x\} = A^T D A x$. From the knowledge of this form, once A is found, D can be found numerically [\[link\]](#).

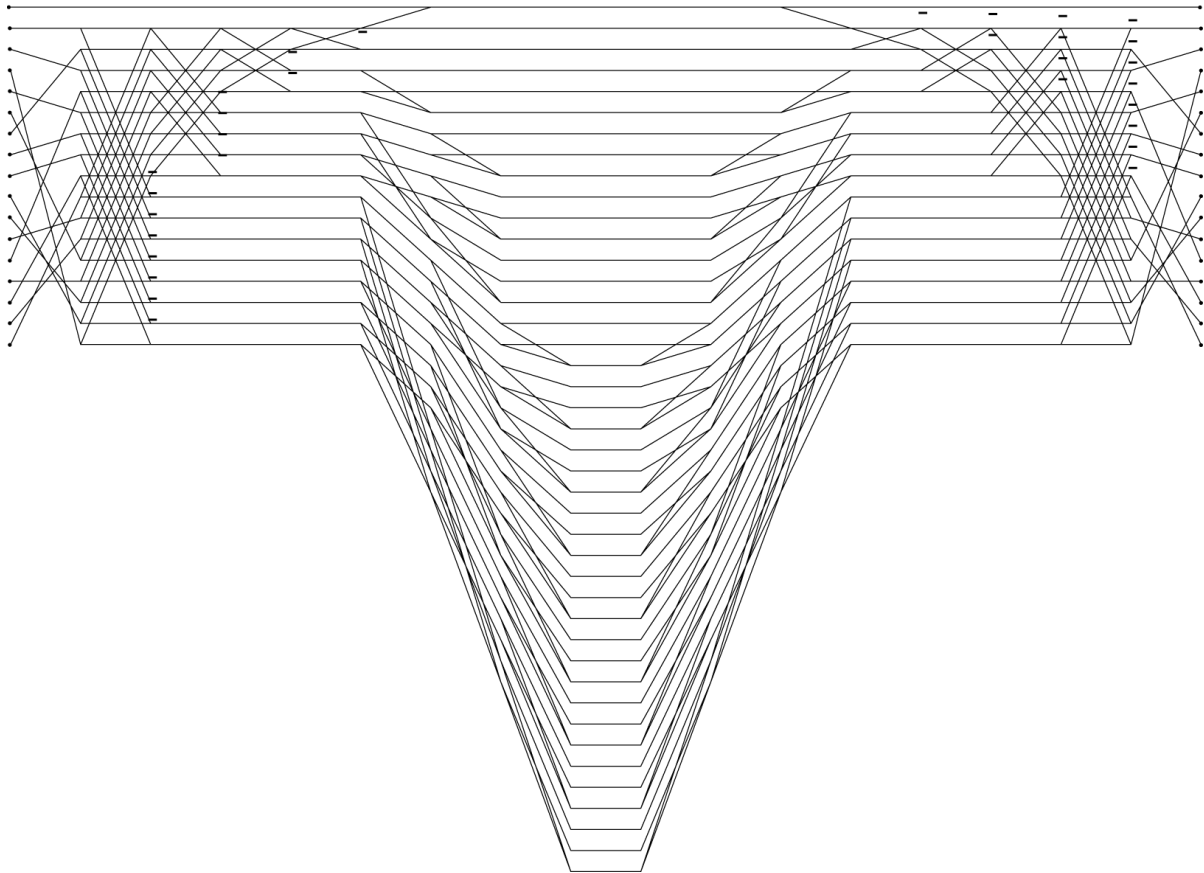
Programming the Design Procedure

Because each of the above steps can be described by matrices, the development of a prime length FFTs is made convenient with the use of a matrix oriented programming language such as Matlab. After specifying the appropriate matrices that describe the desired FFT algorithm, generating code involves compiling the matrices into the desired code for execution.

Each matrix is a section of one stage of the flow graph that corresponds to the DFT program. The four stages are:

1. Permutation Stage: Permutes input and output sequence.
2. Reduction Stage: Reduces the cyclic convolution to smaller polynomial products.
3. Polynomial Product Stage: Performs the polynomial multiplications.
4. Multiplication Stage: Implements the point-by-point multiplication in the bilinear form.

Each of the stages can be clearly seen in the flow graphs for the DFTs. [\[link\]](#) shows the flow graph for a length 17 DFT algorithm that was automatically drawn by the program.

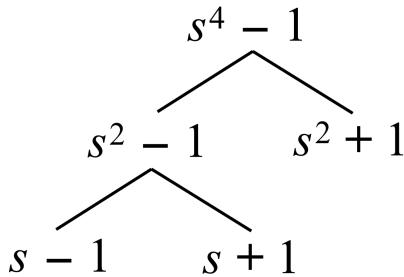


Flowgraph of length-17 DFT

The programs that accomplish this process are written in Matlab and C. Those that compute the appropriate matrices are written in Matlab. These matrices are then stored as two ASCII files, with the dimensions in one and the matrix elements in the second. A C program then reads the files and compiles them to produce the final FFT program in C [\[link\]](#)

The Reduction Stage

The reduction of an N^{th} degree polynomial, $X(s)$, modulo the cyclotomic polynomial factors of $(s^N - 1)$ requires only additions for many N , however, the actual number of additions depends upon the way in which the reduction proceeds. The reduction is most efficiently performed in steps. For example, if $N = 4$ and $((X(s))_{s-1}, ((X(s))_{s+1})$ and $((X(s))_{s^2+1})$ where the double parenthesis denote polynomial reduction modulo $(s - 1)$, $s + 1$, and $s^2 + 1$, then in the first step $((X(s)))_{s^2-1}$, and $((Xs)))_{s^2+1}$ should be computed. In the second step, $((Xs)))_{s-1}$ and $((Xs)))_{s+1}$ can be found by reducing $((X(s)))_{s^2-1}$. This process is described by the diagram in [\[link\]](#).



Factorization of
 $s^4 - 1$ in steps

When N is even, the appropriate first factorization is $(S^{N/2} - 1)(s^{N/2} + 1)$, however, the next appropriate factorization is frequently less obvious. The following procedure has been found to generate a factorization in steps that coincides with the factorization that minimizes the cumulative number of additions incurred by the steps. The prime factors of N are the basis of this procedure and their importance is clear from the useful well-known equation $s^N - 1 = \prod_{n|N} C_n(s)$ where $C_n(s)$ is the n^{th} cyclotomic polynomial.

We first introduce the following two functions defined on the positive integers,

Equation:

$$\psi(N) = \text{the smallest prime factor of } N \text{ for } N > 1$$

and $\psi(1) = 1$.

Suppose $P(s)$ is equal to either $(s^N - 1)$ or an intermediate noncyclotomic polynomial appearing in the factorization process, for example, $(a^2 - 1)$, above. Write $P(s)$ in terms of its cyclotomic factors,

Equation:

$$P(s) = C_{k_1}(s) C_{k_2}(s) \cdots C_{k_L}(s)$$

define the two sets, G and G' , by

Equation:

$$G = \{k_1, \dots, k_L\} \quad \text{and} \quad G' = \{k/\gcd(G) : k \in G\}$$

and define the two integers, t and T , by

Equation:

$$t = \min \{\psi(k) : k \in G, k > 1\} \quad \text{and} \quad T = \max \{nu(k, t) : k \in G\}$$

Then form two new sets,

Equation:

$$A = \{k \in G : T \mid k\} \quad \text{and} \quad B = \{k \in G : T \nmid k\}$$

The factorization of $P(s)$,

Equation:

$$P(s) = \left(\prod_{k \in A} C_k(s) \right) \left(\prod_{k \in B} C_k(s) \right)$$

has been found useful in the procedure for factoring $(s^N - 1)$. This is best illustrated with an example.

Example: $N = 36$

Step 1. Let $P(s) = s^{36} - 1$. Since $P = C_1 C_2 C_3 C_4 C_6 C_9 C_{12} C_{18} C_{36}$

Equation:

$$G = G = \{1, 2, 3, 4, 6, 9, 12, 18, 36\}$$

Equation:

$$t = \min \{2, 3\} = 2$$

Equation:

$$A = \{k \in G : 4|k\} = \{1, 2, 3, 6, 9, 18\}$$

Equation:

$$B = \{k \in G : 4|k\} = \{4, 12, 36\}$$

Hence the factorization of $s^{36} - 1$ into two intermediate polynomials is as expected,

Equation:

$$\prod_{k \in A} C_k(s) = s^{18} - 1, \quad \prod_{k \in B} C_k(s) = s^{18} + 1$$

If a 36th degree polynomial, $X(s)$, is represented by a vector of coefficients, $X = (x_{35}, \dots, x_0)'$, then $((X(s))_{s^{18}-1})$ (represented by X') and $((X(s))_{s^{18}+1})$ (represented by X'') is given by

Equation:

$$test$$

which entails 36 additions.

Step 2. This procedure is repeated with $P(s) = s^{18} - 1$ and $P(s) = s^{18} + 1$. We will just show it for the later. Let $P(s) = s^{18} + 1$. Since $P = C_4 C_{12} C_{36}$

Equation:

$$G = \{4, 12, 36\}, \quad G' = \{l, 3, 9\}$$

Equation:

$$t = \min 3 = 3$$

Equation:

$$T = \max \nu(k, 3) : k \in G = \max l, 3, 9 = 9$$

Equation:

$$A = k \in G : 9|k\} = \{4, 12\}$$

Equation:

$$B = k \in G : 9|k\} = \{36\}$$

This yields the two intermediate polynomials,

Equation:

$$s^6 + 1, \quad \text{and} \quad s^{12} - s^6 + 1$$

In the notation used above,

Equation:

$$\begin{bmatrix} X' \\ X'' \end{bmatrix} = \begin{bmatrix} I_6 & -I_6 & I_6 \\ I_6 & I_6 & \\ -I_6 & & I_6 \end{bmatrix} X$$

entailing 24 additions. Continuing this process results in a factorization in steps

In order to see the number of additions this scheme uses for numbers of the form $N = P - 1$ (which is relevant to prime length FFT algorithms) figure 4 shows the number of additions the reduction process uses when the polynomial $X(s)$ is real.

Figure 4: Number of Additions for Reduction Stage

The Polynomial Product Stage

The iterated convolution algorithm can be used to construct an N point linear convolution algorithm from shorter linear convolution algorithms [\[link\]](#). Suppose the linear convolution y , of the n point vectors x and h (h known) is described by

Equation:

$$y = E_n^T D E_n x$$

where E_n is an “expansion” matrix the elements of which are ± 1 's and 0's and D is an appropriate diagonal matrix. Because the only multiplications in this expression are by the elements of D , the number of multiplications required, $M(n)$, is equal to the number of rows of E_n . The number of additions is denoted by $A(n)$.

Given a matrix E_{n_1} and a matrix E_{n_2} , the iterated algorithm gives a method for combining E_{n_1} and E_{n_2} to construct a valid expansion matrix, E_n , for $N \leq n_1 n_2$. Specifically,

Equation:

$$E_{n_1, n_2} = (I_{M(n_2)} \otimes E_{n_1}) (E_{n_2} \times I_{n_1})$$

The product $n_1 n_2$ may be greater than N , for zeros can be (conceptually) appended to x . The operation count associated with E_{n_1, n_2} is

Equation:

$$A(n_1, n_2) = n! A(n_2) + A(n_1) M n_2$$

Equation:

$$M(n_1, n_2) = M(n_1) M(n_2)$$

Although they are both valid expansion matrices, $E_{n_1, n_2} \neq E_{n_2, n_1}$ and $A_{n_1, n_2} \neq A_{n_2, n_1}$. Because $M_{n_1, n_2} \neq M_{n_2, n_1}$ it is desirable to choose an ordering of factors to minimize the additions incurred by the expansion matrix. The following [\[link\]](#), [\[link\]](#) follows from above.

Multiple Factors

Note that a valid expansion matrix, E_N , can be constructed from E_{n_1, n_2} and E_{n_3} , for $N \leq n_1 n_2 n_3$. In general, any number of factors can be used to create larger expansion matrices. The operation count associated with E_{n_1, n_2, n_3} is

Equation:

$$A(n_1, n_2, n_3) = n_1 n_2 A(n_3) + n_1 A(n_2) M(n_3) + A(n_1) M(n_2) M(n_3)$$

Equation:

$$M(n_1, n_2, n_3) = M(n_1) M(n_2) M(n_3)$$

These equations generalize in the predicted way when more factors are considered. Because the ordering of the factors is relevant in the equation for $A(\cdot)$ but not for $M(\cdot)$, it is again desirable to order the factors to

minimize the number of additions. By exploiting the following property of the expressions for $A(\cdot)$ and $M(\cdot)$, the optimal ordering can be found [\[link\]](#).

reservation of Optimal Ordering. Suppose

$A(n_1, n_2, n_3) \leq \min \{A(n_{k_1}, n_{k_2}, n_{k_3}) : k_1, k_2, k_3 \in \{1, 2, 3\} \text{ and distinct}\}$, then

1. **Equation:**

$$A(n_1, n_2) \leq A(n_2, n_1)$$

2. **Equation:**

$$A(n_2, n_3) \leq A(n_3, n_2)$$

3. **Equation:**

$$A(n_1, n_3) \leq A(n_3, n_1)$$

The generalization of this property to more than two factors reveals that an optimal ordering of $\{n_1, \dots, n_{L-i}\}$ is preserved in an optimal ordering of $\{n_1, \dots, n_L\}$. Therefore, if (n_1, \dots, n_L) is an optimal ordering of $\{n_1, \dots, n_L\}$, then (n_k, n_{k+1}) is an optimal ordering of $\{n_k, n_{k+1}\}$ and consequently

Equation:

$$\frac{A(n_k)}{M(n_k) - n_k} \leq \frac{A(n_{k+1})}{M(n_{k+1}) - n_{k+1}}$$

for all $k = 1, 2, \dots, L - 1$.

This immediately suggests that an optimal ordering of $\{n_1, \dots, n_L\}$ is one for which

Equation:

$$\frac{A(n_1)}{M(n_1) - n_1} \cdots \frac{A(n_L)}{M(n_L) - n_L}$$

is nondecreasing. Hence, ordering the factors, $\{n_1, \dots, n_L\}$, to minimize the number of additions incurred by E_{n_1, \dots, n_L} simply involves computing the appropriate ratios.

Discussion and Conclusion

We have designed prime length FFTs up to length 53 that are as good as the previous designs that only went up to 19. Table 1 gives the operation counts for the new and previously designed modules, assuming complex inputs.

It is interesting to note that the operation counts depend on the factorability of $P - 1$. The primes 11, 23, and 47 are all of the form $1 + 2P_1$ making the design of efficient FFTs for these lengths more difficult.

Further deviations from the original Winograd approach than we have made could prove useful for longer lengths. We investigated, for example, the use of twiddle factors at appropriate points in the decomposition stage; these can sometimes be used to divide the cyclic convolution into smaller convolutions. Their use means, however, that the 'center*' multiplications would no longer be by purely real or imaginary numbers.

N	Mult	Adds
7	16	72
11	40	168

13	40	188
17	82	274
19	88	360
23	174	672
29	190	766
31	160	984
37	220	920
41	282	1140
43	304	1416
47	640	2088
53	556	2038

Operation counts for prime length DFTs

The approach in writing a program that writes another program is a valuable one for several reasons. Programming the design process for the design of prime length FFTs has the advantages of being practical, error-free, and flexible. The flexibility is important because it allows for modification and experimentation with different algorithmic ideas. Above all, it has allowed longer DFTs to be reliably designed.

More details on the generation of programs for prime length FFTs can be found in the 1993 Technical Report.

DFT and FFT: An Algebraic View

by Markus Pueschel, Carnegie Mellon University

In infinite, or non-periodic, discrete-time signal processing, there is a strong connection between the z -transform, Laurent series, convolution, and the discrete-time Fourier transform (DTFT) [\[link\]](#). As one may expect, a similar connection exists for the DFT but bears surprises. Namely, it turns out that the proper framework for the DFT requires modulo operations of polynomials, which means working with so-called polynomial algebras [\[link\]](#). Associated with polynomial algebras is the Chinese remainder theorem, which describes the DFT algebraically and can be used as a tool to concisely derive various FFTs as well as convolution algorithms [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#) (see also [Winograd's Short DFT Algorithms](#)). The polynomial algebra framework was fully developed for signal processing as part of the algebraic signal processing theory (ASP). ASP identifies the structure underlying many transforms used in signal processing, provides deep insight into their properties, and enables the derivation of their fast algorithms [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#). Here we focus on the algebraic description of the DFT and on the algebraic derivation of the general-radix Cooley-Tukey FFT from [Factoring the Signal Processing Operators](#). The derivation will make use of and extend the [Polynomial Description of Signals](#). We start with motivating the appearance of modulo operations.

The z -transform associates with infinite discrete signals $X = (\dots, x(-1), x(0), x(1), \dots)$ a Laurent series:

Equation:

$$X \mapsto X(s) = \sum_{n \in \mathbb{Z}} x(n) s^n.$$

Here we used $s = z^{-1}$ to simplify the notation in the following. The DTFT of X is the evaluation of $X(s)$ on the unit circle

Equation:

$$X(e^{-j\omega}), \quad -\pi < \omega \leq \pi.$$

Finally, filtering or (linear) convolution is simply the multiplication of Laurent series,

Equation:

$$H * X \leftrightarrow H(s)X(s).$$

For finite signals $X = (x(0), \dots, x(N-1))$ one expects that the equivalent of [\[link\]](#) becomes a mapping to polynomials of degree $N-1$,

Equation:

$$X \mapsto X(s) = \sum_{n=0}^{N-1} x(n) s^n,$$

and that the DFT is an evaluation of these polynomials. Indeed, the definition of the DFT in [Winograd's Short DFT Algorithms](#) shows that

Equation:

$$C(k) = X(W_N^k) = X\left(e^{-j\frac{2\pi k}{N}}\right), \quad 0 \leq k < N,$$

i.e., the DFT computes the evaluations of the polynomial $X(s)$ at the n th roots of unity.

The problem arises with the equivalent of [\[link\]](#), since the multiplication $H(s)X(s)$ of two polynomials of degree $N - 1$ yields one of degree $2N - 2$. Also, it does not coincide with the circular convolution known to be associated with the DFT. The solution to both problems is to reduce the product modulo $s^n - 1$:

Equation:

$$H^*_{\text{circ}} X \leftrightarrow H(s)X(s) \bmod (s^n - 1).$$

Concept	Infinite Time	Finite Time
Signal	$X(s) = \sum_{n \in \mathbb{Z}} x(n)s^n$	$\sum_{n=0}^{N-1} x(n)s^n$
Filter	$H(s) = \sum_{n \in \mathbb{Z}} h(n)s^n$	$\sum_{n=0}^{N-1} h(n)s^n$
Convolution	$H(s)X(s)$	$H(s)X(s) \bmod (s^n - 1)$
Fourier transform	DTFT: $X(e^{-j\omega})$, $-\pi < \omega \leq \pi$	DFT: $X\left(e^{-j\frac{2\pi k}{n}}\right)$, $0 \leq k < n$

Infinite and finite discrete time signal processing.

The resulting polynomial then has again degree $N - 1$ and this form of convolution becomes equivalent to circular convolution of the polynomial coefficients. We also observe that the evaluation points in [\[link\]](#) are precisely the roots of $s^n - 1$. This connection will become clear in this chapter.

The discussion is summarized in [\[link\]](#).

The proper framework to describe the multiplication of polynomials modulo a fixed polynomial are polynomial algebras. Together with the Chinese remainder theorem, they provide the theoretical underpinning for the DFT and the Cooley-Tukey FFT.

In this chapter, the DFT will naturally arise as a linear mapping with respect to chosen bases, i.e., as a matrix. Indeed, the definition shows that if all input and outputs are collected into vectors

$X = (X(0), \dots, X(N-1))$ and $C = (C(0), \dots, C(N-1))$, then [Winograd's Short DFT Algorithms](#) is equivalent to

Equation:

$$C = \text{DFT}_N X,$$

where

Equation:

$$\text{DFT}_N = [W_N^{kn}]_{0 \leq k, n < N}.$$

The matrix point of view is adopted in the FFT books [\[link\]](#), [\[link\]](#).

Polynomial Algebras and the DFT

In this section we introduce polynomial algebras and explain how they are associated to transforms. Then we identify this connection for the DFT. Later we use polynomial algebras to derive the Cooley-Tukey FFT.

For further background on the mathematics in this section and polynomial algebras in particular, we refer to [\[link\]](#).

Polynomial Algebra

An algebra \mathcal{A} is a vector space that also provides a multiplication of its elements such that the distributivity law holds (see [\[link\]](#) for a complete definition). Examples include the sets of complex or real numbers \mathbb{C} or \mathbb{R} , and the sets of complex or real polynomials in the variable s : $\mathbb{C}[s]$ or $\mathbb{R}[s]$.

The key player in this chapter is the **polynomial algebra**. Given a fixed polynomial $P(s)$ of degree $\deg(P) = N$, we define a polynomial algebra as the set

Equation:

$$\mathbb{C}[s]/P(s) = \{X(s) \mid \deg(X) < \deg(P)\}$$

of polynomials of degree smaller than N with addition and multiplication modulo P . Viewed as a vector space, $\mathbb{C}[s]/P(s)$ hence has dimension N .

Every polynomial $X(s) \in \mathbb{C}[s]$ is reduced to a unique polynomial $R(s)$ modulo $P(s)$ of degree smaller than N . $R(s)$ is computed using division with rest, namely

Equation:

$$X(s) = Q(s)P(s) + R(s), \quad \deg(R) < \deg(P).$$

Regarding this equation modulo P , $P(s)$ becomes zero, and we get

Equation:

$$X(s) \equiv R(s) \pmod{P(s)}.$$

We read this equation as “ $X(s)$ is congruent (or equal) $R(s)$ modulo $P(s)$.” We will also write $X(s) \pmod{P(s)}$ to denote that $X(s)$ is reduced modulo $P(s)$. Obviously,

Equation:

$$P(s) \equiv 0 \pmod{P(s)}.$$

As a simple example we consider $\mathcal{A} = \mathbb{C}[s]/(s^2 - 1)$, which has dimension 2. A possible basis is $b = (1, s)$. In \mathcal{A} , for example, $s \cdot (s + 1) = s^2 + s \equiv s + 1 \pmod{(s^2 - 1)}$, obtained through division with rest

Equation:

$$s^2 + s = 1 \cdot (s^2 - 1) + (s + 1)$$

or simply by replacing s^2 with 1 (since $s^2 - 1 = 0$ implies $s^2 = 1$).

Chinese Remainder Theorem (CRT)

Assume $P(s) = Q(s)R(s)$ factors into two coprime (no common factors) polynomials Q and R . Then the Chinese remainder theorem (CRT) for polynomials is the linear mapping [\[footnote\]](#). More precisely, isomorphism of algebras or isomorphism of \mathcal{A} -modules.

Equation:

$$\begin{aligned} \Delta : \mathbb{C}[s]/P(s) &\rightarrow \mathbb{C}[s]/Q(s) \oplus \mathbb{C}[s]/R(s), \\ X(s) &\mapsto (X(s) \pmod{Q(s)}, X(s) \pmod{R(s)}). \end{aligned}$$

Here, \oplus is the Cartesian product of vector spaces with elementwise operation (also called outer direct sum). In words, the CRT asserts that computing (addition, multiplication, scalar multiplication) in $\mathbb{C}[s]/P(s)$ is equivalent to computing in parallel in $\mathbb{C}[s]/Q(s)$ and $\mathbb{C}[s]/R(s)$.

If we choose bases b, c, d in the three polynomial algebras, then Δ can be expressed as a matrix. As usual with linear mappings, this matrix is obtained by mapping every element of b with Δ , expressing it in the concatenation $c \cup d$ of the bases c and d , and writing the results into the columns of the matrix.

As an example, we consider again the polynomial $P(s) = s^2 - 1 = (s - 1)(s + 1)$ and the CRT decomposition

Equation:

$$\Delta : \mathbb{C}[s]/(s^2 - 1) \rightarrow \mathbb{C}[s]/(s - 1) \oplus \mathbb{C}[s]/(s + 1).$$

As bases, we choose $b = (1, x)$, $c = (1)$, $d = (1)$. $\Delta(1) = (1, 1)$ with the same coordinate vector in $c \cup d = (1, 1)$. Further, because of $x \equiv 1 \pmod{(x-1)}$ and $x \equiv -1 \pmod{(x+1)}$, $\Delta(x) = (x, x) \equiv (1, -1)$ with the same coordinate vector. Thus, Δ in matrix form is the so-called butterfly matrix, which is a DFT of size 2: $\text{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.

Polynomial Transforms

Assume $P(s) \in \mathbb{C}[s]$ has pairwise distinct zeros $\alpha = (\alpha_0, \dots, \alpha_{N-1})$. Then the CRT can be used to completely decompose $\mathbb{C}[s]/P(s)$ into its **spectrum**:

Equation:

$$\begin{aligned} \Delta : \mathbb{C}[s]/P(s) &\rightarrow \mathbb{C}[s]/(s - \alpha_0) \oplus \dots \oplus \mathbb{C}[s]/(s - \alpha_{N-1}), \\ X(s) &\mapsto (X(s) \bmod (s - \alpha_0), \dots, X(s) \bmod (s - \alpha_{N-1})) \\ &= (s(\alpha_0), \dots, s(\alpha_{N-1})). \end{aligned}$$

If we choose a basis $b = (P_0(s), \dots, P_{N-1}(s))$ in $\mathbb{C}[s]/P(s)$ and bases $b_i = (1)$ in each $\mathbb{C}[s]/(s - \alpha_i)$, then Δ , as a linear mapping, is represented by a matrix. The matrix is obtained by mapping every basis element P_n , $0 \leq n < N$, and collecting the results in the columns of the matrix. The result is

Equation:

$$\mathcal{P}_{b,\alpha} = [P_n(\alpha_k)]_{0 \leq k, n < N}$$

and is called the **polynomial transform** for $\mathcal{A} = \mathbb{C}[s]/P(s)$ with basis b .

If, in general, we choose $b_i = (\beta_i)$ as spectral basis, then the matrix corresponding to the decomposition [\[link\]](#) is the **scaled polynomial transform**

Equation:

$$\text{diag}_{0 \leq k < N} (1/\beta_n) \mathcal{P}_{b,\alpha},$$

where $\text{diag}_{0 \leq n < N} (\gamma_n)$ denotes a diagonal matrix with diagonal entries γ_n .

We jointly refer to polynomial transforms, scaled or not, as Fourier transforms.

DFT as a Polynomial Transform

We show that the DFT_N is a polynomial transform for $\mathcal{A} = \mathbb{C}[s]/(s^N - 1)$ with basis $b = (1, s, \dots, s^{N-1})$. Namely,

Equation:

$$s^N - 1 = \prod_{0 \leq k < N} (x - W_N^k),$$

which means that Δ takes the form

Equation:

$$\begin{aligned} \Delta : \mathbb{C}[s]/(s^N - 1) &\rightarrow \mathbb{C}[s]/(s - W_N^0) \oplus \cdots \oplus \mathbb{C}[s]/(s - W_N^{N-1}), \\ X(s) &\mapsto (X(s) \bmod (s - W_N^0), \dots, X(s) \bmod (s - W_N^{N-1})) \\ &= (X(W_N^0), \dots, X(W_N^{N-1})). \end{aligned}$$

The associated polynomial transform hence becomes

Equation:

$$\mathcal{P}_{b,\alpha} = [W_N^{kn}]_{0 \leq k, n < N} = \text{DFT}_N.$$

This interpretation of the DFT has been known at least since [\[link\]](#), [\[link\]](#) and clarifies the connection between the evaluation points in [\[link\]](#) and the circular convolution in [\[link\]](#).

In [\[link\]](#), DFTs of types 1–4 are defined, with type 1 being the standard DFT. In the algebraic framework, type 3 is obtained by choosing $\mathcal{A} = \mathbb{C}[s]/(s^N + 1)$ as algebra with the same basis as before:

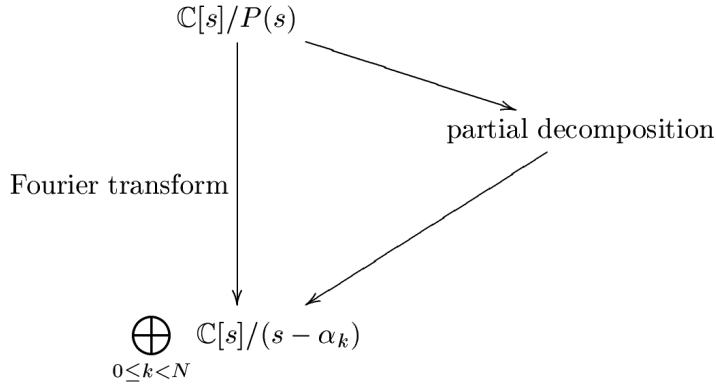
Equation:

$$\mathcal{P}_{b,\alpha} = [W_N^{(k+1/2)n}]_{0 \leq k, n < N} = \text{DFT-3}_N,$$

The DFTs of type 2 and 4 are scaled polynomial transforms [\[link\]](#).

Algebraic Derivation of the Cooley-Tukey FFT

Knowing the polynomial algebra underlying the DFT enables us to derive the Cooley-Tukey FFT **algebraically**. This means that instead of manipulating the DFT definition, we manipulate the polynomial algebra $\mathbb{C}[s]/(s^N - 1)$. The basic idea is intuitive. We showed that the DFT is the matrix representation of the complete decomposition [\[link\]](#). The Cooley-Tukey FFT is now derived by performing this decomposition **in steps** as shown in [\[link\]](#). Each step yields a sparse matrix; hence, the DFT_N is factorized into a product of sparse matrices, which will be the matrix representation of the Cooley-Tukey FFT.



Basic idea behind the algebraic derivation of Cooley-Tukey type algorithms

This stepwise decomposition can be formulated generically for polynomial transforms [\[link\]](#), [\[link\]](#). Here, we consider only the DFT.

We first introduce the matrix notation we will use and in particular the Kronecker product formalism that became mainstream for FFTs in [\[link\]](#), [\[link\]](#).

Then we first derive the radix-2 FFT using a **factorization** of $s^N - 1$. Subsequently, we obtain the general-radix FFT using a **decomposition** of $s^N - 1$.

Matrix Notation

We denote the $N \times N$ identity matrix with I_N , and diagonal matrices with **Equation:**

$$\text{diag}_{0 \leq k < N} (\gamma_k) = \begin{bmatrix} \gamma_0 & & \\ & \ddots & \\ & & \gamma_{N-1} \end{bmatrix}.$$

The $N \times N$ **stride permutation** matrix is defined for $N = KM$ by the permutation **Equation:**

$$L_M^N : iK + j \mapsto jM + i$$

for $0 \leq i < K$, $0 \leq j < M$. This definition shows that L_M^N transposes a $K \times M$ matrix stored in row-major order. Alternatively, we can write

Equation:

$$L_M^N : i \mapsto iM \bmod N - 1, \text{ for } 0 \leq i < N - 1, \quad N - 1 \mapsto N - 1.$$

For example (\cdot means 0),

Equation:

$$L_2^6 = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{bmatrix}.$$

$L_{N/2}^N$ is sometimes called the perfect shuffle.

Further, we use matrix operators; namely the direct sum

Equation:

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}$$

and the Kronecker or tensor product

Equation:

$$A \otimes B = [a_{k,\ell} B]_{k,\ell}, \quad \text{for } A = [a_{k,\ell}].$$

In particular,

Equation:

$$I_n \otimes A = A \oplus \cdots \oplus A = \begin{bmatrix} A & & \\ & \ddots & \\ & & A \end{bmatrix}$$

is block-diagonal.

We may also construct a larger matrix as a matrix of matrices, e.g.,

Equation:

$$\begin{bmatrix} A & B \\ B & A \end{bmatrix}.$$

If an algorithm for a transform is given as a product of sparse matrices built from the constructs above, then an algorithm for the transpose or inverse of the transform can be readily derived using mathematical properties including

Equation:

$$\begin{aligned}(AB)^T &= B^T A^T, & (AB)^{-1} &= B^{-1} A^{-1}, \\ (A \oplus B)^T &= A^T \oplus B^T, & (A \oplus B)^{-1} &= A^{-1} \oplus B^{-1}, \\ (A \otimes B)^T &= A^T \otimes B^T, & (A \otimes B)^{-1} &= A^{-1} \otimes B^{-1}.\end{aligned}$$

Permutation matrices are orthogonal, i.e., $P^T = P^{-1}$. The transposition or inversion of diagonal matrices is obvious.

Radix-2 FFT

The DFT decomposes $\mathcal{A} = \mathbb{C}[s]/(s^N - 1)$ with basis $b = (1, s, \dots, s^{N-1})$ as shown in [\[link\]](#). We assume $N = 2M$. Then

Equation:

$$s^{2M} - 1 = (s^M - 1)(s^M + 1)$$

factors and we can apply the CRT in the following steps:

Equation:

$$\begin{aligned}& \mathbb{C}[s]/(s^N - 1) \\ \rightarrow & \mathbb{C}[s]/(s^M - 1) \oplus \mathbb{C}[s]/(s^M + 1)\end{aligned}$$

Equation:

$$\rightarrow \bigoplus_{0 \leq i < M} \mathbb{C}[s]/(x - W_N^{2i}) \oplus \bigoplus_{0 \leq i < M} \mathbb{C}[s]/(x - W_M^{2i+1})$$

Equation:

$$\rightarrow \bigoplus_{0 \leq i < N} \mathbb{C}[s]/(x - W_N^i).$$

As bases in the smaller algebras $\mathbb{C}[s]/(s^M - 1)$ and $\mathbb{C}[s]/(s^M + 1)$, we choose $c = d = (1, s, \dots, s^{M-1})$. The derivation of an algorithm for DFT_N based on [\[link\]](#)-[\[link\]](#) is now completely mechanical by reading off the matrix for each of the three decomposition steps. The product of these matrices is equal to the DFT_N .

First, we derive the base change matrix B corresponding to [\[link\]](#). To do so, we have to express the base elements $s^n \in b$ in the basis $c \cup d$; the coordinate vectors are the columns of B . For $0 \leq n < M$, s^n is actually contained in c and d , so the first M columns of B are

Equation:

$$B = \begin{bmatrix} I_M & * \\ I_M & * \end{bmatrix},$$

where the entries $*$ are determined next. For the base elements s^{M+n} , $0 \leq n < M$, we have

Equation:

$$\begin{aligned} s^{M+n} &\equiv s^n \pmod{(s^M - 1)}, \\ s^{M+n} &\equiv -s^n \pmod{(s^M + 1)}, \end{aligned}$$

which yields the final result

Equation:

$$B = \begin{bmatrix} I_M & I_M \\ I_M & -I_M \end{bmatrix} = \text{DFT}_2 \otimes I_M.$$

Next, we consider step [\[link\]](#). $\mathbb{C}[s]/(s^M - 1)$ is decomposed by DFT_M and $\mathbb{C}[s]/(s^M + 1)$ by DFT_{-3_M} in [\[link\]](#).

Finally, the permutation in step [\[link\]](#) is the perfect shuffle L_M^N , which interleaves the even and odd spectral components (even and odd exponents of W_N).

The final algorithm obtained is

Equation:

$$\text{DFT}_{2M} = L_M^N (\text{DFT}_M \oplus \text{DFT}_{-3_M}) (\text{DFT}_2 \otimes I_M).$$

To obtain a better known form, we use $\text{DFT}_{-3_M} = \text{DFT}_M D_M$, with $D_M = \text{diag}_{0 \leq i < M} (W_N^i)$, which is evident from [\[link\]](#). It yields

Equation:

$$\begin{aligned} \text{DFT}_{2M} &= L_M^N (\text{DFT}_M \oplus \text{DFT}_M D_M) (\text{DFT}_2 \otimes I_M) \\ &= L_M^N (I_2 \otimes \text{DFT}_M) (I_M \oplus D_M) (\text{DFT}_2 \otimes I_M). \end{aligned}$$

The last expression is the radix-2 decimation-in-frequency Cooley-Tukey FFT. The corresponding decimation-in-time version is obtained by transposition using [\[link\]](#) and the symmetry of the DFT:

Equation:

$$\text{DFT}_{2M} = (\text{DFT}_2 \otimes I_M) (I_M \oplus D_M) (I_2 \otimes \text{DFT}_M) L_2^N.$$

The entries of the diagonal matrix $I_M \oplus D_M$ are commonly called **twiddle factors**.

The above method for deriving DFT algorithms is used extensively in [\[link\]](#).

General-radix FFT

To algebraically derive the general-radix FFT, we use the **decomposition property** of $s^N - 1$. Namely, if $N = KM$ then

Equation:

$$s^N - 1 = (s^M)^K - 1.$$

Decomposition means that the polynomial is written as the composition of two polynomials: here, s^M is inserted into $s^K - 1$. Note that this is a special property: most polynomials do not decompose.

Based on this polynomial decomposition, we obtain the following stepwise decomposition of $\mathbb{C}[s]/(s^N - 1)$, which is more general than the previous one in [\[link\]](#)–[\[link\]](#). The basic idea is to first decompose with respect to the outer polynomial $t^K - 1$, $t = s^M$, and then completely [\[link\]](#):

Equation:

$$\begin{aligned} \mathbb{C}[s]/(s^N - 1) &= \mathbb{C}[x]/((s^M)^K - 1) \\ \rightarrow \bigoplus_{0 \leq i < K} \mathbb{C}[s]/(s^M - W_K^i) \end{aligned}$$

Equation:

$$\rightarrow \bigoplus_{0 \leq i < K} \bigoplus_{0 \leq j < M} \mathbb{C}[s]/(x - W_N^{jK+i})$$

Equation:

$$\rightarrow \bigoplus_{0 \leq i < N} \mathbb{C}[s]/(x - W_N^i).$$

As bases in the smaller algebras $\mathbb{C}[s]/(s^M - W_K^i)$ we choose $c_i = (1, s, \dots, s^{M-1})$. As before, the derivation is completely mechanical from here: only the three matrices corresponding to [\[link\]](#)–[\[link\]](#) have to be read off.

The first decomposition step requires us to compute $s^n \bmod (s^M - W_K^i)$, $0 \leq n < N$. To do so, we decompose the index n as $n = \ell M + m$ and compute

Equation:

$$s^n = s^{\ell M + m} = (s^M)^\ell s^m \equiv W_K^{\ell m} s^m \bmod (s^M - W_K^i).$$

This shows that the matrix for [\[link\]](#) is given by $\text{DFT}_K \otimes I_M$.

In step [\[link\]](#), each $\mathbb{C}[s]/(s^M - W_K^i)$ is completely decomposed by its polynomial transform
Equation:

$$\text{DFT}_M(i, K) = \text{DFT}_M \cdot \text{diag}_{0 \leq i < M} (W_N^{ij}).$$

At this point, $\mathbb{C}[s]/(s^N - 1)$ is completely decomposed, but the spectrum is ordered according to $jK + i$, $0 \leq i < M$, $0 \leq j < K$ (j runs faster). The desired order is $iM + j$.

Thus, in step [\[link\]](#), we need to apply the permutation $jK + i \mapsto iM + j$, which is exactly the stride permutation L_M^N in [\[link\]](#).

In summary, we obtain the Cooley-Tukey decimation-in-frequency FFT with arbitrary radix:
Equation:

$$\begin{aligned} & L_M^N \left(\bigoplus_{0 \leq i < K} \text{DFT}_M \cdot \text{diag}_{j=0}^{M-1} (W_N^{ij}) \right) (\text{DFT}_k \otimes I_M) \\ &= L_M^N (I_K \otimes \text{DFT}_M) T_M^N (\text{DFT}_k \otimes I_M). \end{aligned}$$

The matrix T_M^N is diagonal and usually called the **twiddle matrix**. Transposition using [\[link\]](#) yields the corresponding decimation-in-time version:

Equation:

$$(\text{DFT}_k \otimes I_M) T_M^N (I_K \otimes \text{DFT}_M) L_K^N.$$

Discussion and Further Reading

This chapter only scratches the surface of the connection between algebra and the DFT or signal processing in general. We provide a few references for further reading.

Algebraic Derivation of Transform Algorithms

As mentioned before, the use of polynomial algebras and the CRT underlies much of the early work on FFTs and convolution algorithms [\[link\]](#), [\[link\]](#), [\[link\]](#). For example, Winograd's work on FFTs minimizes the number of non-rational multiplications. This and his work on complexity theory in general makes heavy use of polynomial algebras [\[link\]](#), [\[link\]](#), [\[link\]](#) (see Chapter [Winograd's Short DFT Algorithms](#) for more information and references). See [\[link\]](#) for a broad treatment of algebraic complexity theory.

Since $\mathbb{C}[x]/(s^N - 1) = \mathbb{C}[C_N]$ can be viewed a group algebra for the cyclic group, the methods shown in this chapter can be translated into the context of group representation theory. For example, [\[link\]](#) derives the general-radix FFT using group theory and also uses already the Kronecker product formalism. So does Beth and started the area of FFTs for more general groups [\[link\]](#), [\[link\]](#). However, Fourier transforms for groups have found only sporadic applications [\[link\]](#). Along a

related line of work, [\[link\]](#) shows that using group theory it is possible that to discover and generate certain algorithms for trigonometric transforms, such as discrete cosine transforms (DCTs), automatically using a computer program.

More recently, the polynomial algebra framework was extended to include most trigonometric transforms used in signal processing [\[link\]](#), [\[link\]](#), namely, besides the DFT, the discrete cosine and sine transforms and various real DFTs including the discrete Hartley transform. It turns out that the same techniques shown in this chapter can then be applied to derive, explain, and classify most of the known algorithms for these transforms and even obtain a large class of new algorithms including general-radix algorithms for the discrete cosine and sine transforms (DCTs/DSTs) [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#).

This latter line of work is part of the algebraic signal processing theory briefly discussed next.

Algebraic Signal Processing Theory

The algebraic properties of transforms used in the above work on algorithm derivation hints at a connection between algebra and (linear) signal processing itself. This is indeed the case and was fully developed in a recent body of work called algebraic signal processing theory (ASP). The foundation of ASP is developed in [\[link\]](#), [\[link\]](#), [\[link\]](#).

ASP first identifies the algebraic structure of (linear) signal processing: the common assumptions on available operations for filters and signals make the set of filters an **algebra** \mathcal{A} and the set of signals an associated \mathcal{A} -module \mathcal{M} . ASP then builds a signal processing theory formally from the axiomatic definition of a **signal model**: a triple $(\mathcal{A}, \mathcal{M}, \Phi)$, where Φ generalizes the idea of the z -transform to mappings from vector spaces of signal values to \mathcal{M} . If a signal model is given, other concepts, such as spectrum, Fourier transform, frequency response are automatically defined but take different forms for different models. For example, infinite and finite time as discussed in [\[link\]](#) are two examples of signal models. Their complete definition is provided in [\[link\]](#) and identifies the proper notion of a finite z -transform as a mapping $\mathbb{C}^n \rightarrow \mathbb{C}[s]/(s^n - 1)$.

Signal model	Infinite time	Finite time
\mathcal{A}	$\left\{ \sum_{n \in \mathbb{Z}} H(n) s^n \mid (\dots, H(-1), H(0), H(1), \dots) \in \ell^1(\mathbb{Z}) \right\}$	$\mathbb{C}[x]/(s^n - 1)$
\mathcal{M}	$\left\{ \sum_{n \in \mathbb{Z}} X(n) s^n \mid (\dots, X(-1), X(0), X(1), \dots) \in \ell^2(\mathbb{Z}) \right\}$	$\mathbb{C}[s]/(s^n - 1)$
Φ	$\Phi : \ell^2(\mathbb{Z}) \rightarrow \mathcal{M}$	$\Phi : \mathbb{C}^n \rightarrow \mathcal{M}$

	defined in [link]	defined in [link]
--	-----------------------------------	-----------------------------------

Infinite and finite time models as defined in ASP.

ASP shows that many signal models are in principle possible, each with its own notion of filtering and Fourier transform. Those that support shift-invariance have commutative algebras. Since finite-dimensional commutative algebras are precisely polynomial algebras, their appearance in signal processing is explained. For example, ASP identifies the polynomial algebras underlying the DCTs and DSTs, which hence become Fourier transforms in the ASP sense. The signal models are called finite **space** models since they support signal processing based on an undirected shift operator, different from the directed time shift. Many more insights are provided by ASP including the need for and choices in choosing boundary conditions, properties of transforms, techniques for deriving new signal models, and the concise derivation of algorithms mentioned before.

The Cooley-Tukey Fast Fourier Transform Algorithm

The publication by Cooley and Tukey [\[link\]](#) in 1965 of an efficient algorithm for the calculation of the DFT was a major turning point in the development of digital signal processing. During the five or so years that followed, various extensions and modifications were made to the original algorithm [\[link\]](#). By the early 1970's the practical programs were basically in the form used today. The standard development presented in [\[link\]](#), [\[link\]](#), [\[link\]](#) shows how the DFT of a length- N sequence can be simply calculated from the two length- $N/2$ DFT's of the even index terms and the odd index terms. This is then applied to the two half-length DFT's to give four quarter-length DFT's, and repeated until N scalars are left which are the DFT values. Because of alternately taking the even and odd index terms, two forms of the resulting programs are called decimation-in-time and decimation-in-frequency. For a length of 2^M , the dividing process is repeated $M = \log_2 N$ times and requires N multiplications each time. This gives the famous formula for the computational complexity of the FFT of $N \log_2 N$ which was derived in [Multidimensional Index Mapping: Equation 34](#).

Although the decimation methods are straightforward and easy to understand, they do not generalize well. For that reason it will be assumed that the reader is familiar with that description and this chapter will develop the FFT using the index map from [Multidimensional Index Mapping](#).

The Cooley-Tukey FFT always uses the Type 2 index map from [Multidimensional Index Mapping: Equation 11](#). This is necessary for the most popular forms that have $N = R^M$, but is also used even when the factors are relatively prime and a Type 1 map could be used. The time and frequency maps from [Multidimensional Index Mapping: Equation 6](#) and [Multidimensional Index Mapping: Equation 12](#) are **Equation:**

$$n = ((K_1 n_1 + K_2 n_2))_N$$

Equation:

$$k = ((K_3 k_1 + K_4 k_2))_N$$

Type-2 conditions [Multidimensional Index Mapping: Equation 8](#) and [Multidimensional Index Mapping: Equation 11](#) become

Equation:

$$K_1 = aN_2 \quad \text{or} \quad K_2 = bN_1 \quad \text{but not both}$$

and

Equation:

$$K_3 = cN_2 \quad \text{or} \quad K_4 = dN_1 \quad \text{but not both}$$

The row and column calculations in [Multidimensional Index Mapping: Equation 15](#) are uncoupled by [Multidimensional Index Mapping: Equation 16](#) which for this case are

Equation:

$$((K_1 K_4))_N = 0 \quad \text{or} \quad ((K_2 K_3))_N = 0 \quad \text{but not both}$$

To make each short sum a DFT, the K_i must satisfy

Equation:

$$((K_1 K_3))_N = N_2 \quad \text{and} \quad ((K_2 K_4))_N = N_1$$

In order to have the smallest values for K_i the constants in [\[link\]](#) are chosen to be
Equation:

$$a = d = K_2 = K_3 = 1$$

which makes the index maps of [\[link\]](#) become
Equation:

$$n = N_2 n_1 + n_2$$

Equation:

$$k = k_1 + N_1 k_2$$

These index maps are all evaluated modulo N , but in [\[link\]](#), explicit reduction is not necessary since n never exceeds N . The reduction notation will be omitted for clarity. From [Multidimensional Index Mapping: Equation 15](#) and example [Multidimensional Index Mapping: Equation 19](#), the DFT is
Equation:

$$X = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x \, W_{N_1}^{n_1 k_1} \, W_N^{n_2 k_1} \, W_{N_2}^{n_2 k_2}$$

This map of [\[link\]](#) and the form of the DFT in [\[link\]](#) are the fundamentals of the Cooley-Tukey FFT.

The order of the summations using the Type 2 map in [\[link\]](#) cannot be reversed as it can with the Type-1 map. This is because of the W_N terms, the twiddle factors.

Turning [\[link\]](#) into an efficient program requires some care. From [Multidimensional Index Mapping: Efficiencies Resulting from Index Mapping with the DFT](#) we know that all the factors should be equal. If $N = R^M$, with R called the radix, N_1 is first set equal to R and N_2 is then necessarily R^{M-1} . Consider n_1 to be the index along the rows and n_2 along the columns. The inner sum of [\[link\]](#) over n_1 represents a length- N_1 DFT for each value of n_2 . These N_2 length- N_1 DFT's are the DFT's of the rows of the $x(n_1, n_2)$ array. The resulting array of row DFT's is multiplied by an array of twiddle factors which are the W_N terms in [\[link\]](#). The twiddle-factor array for a length-8 radix-2 FFT is

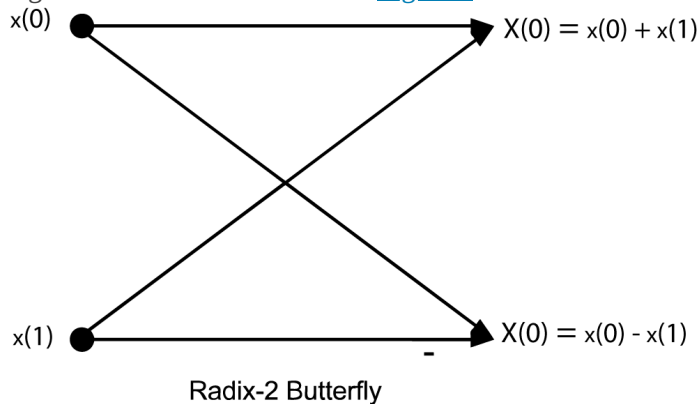
Equation:

$$TF : \quad W_8^{n_2 k_1} = \begin{matrix} W^0 & W^0 & 1 & 1 \\ W^0 & W^1 & 1 & W \\ W^0 & W^2 & 1 & -j \\ W^0 & W^3 & 1 & -jW \end{matrix} =$$

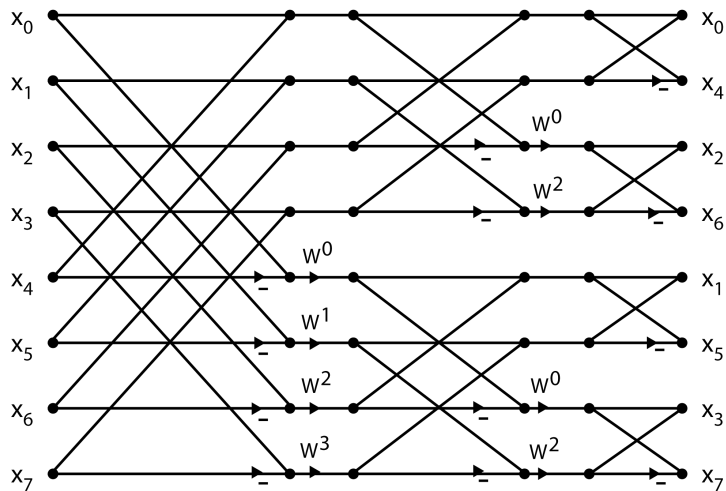
The twiddle factor array will always have unity in the first row and first column.

To complete [\[link\]](#) at this point, after the row DFT's are multiplied by the TF array, the N_1 length- N_2 DFT's of the columns are calculated. However, since the columns DFT's are of length R^{M-1} , they can be posed as a R^{M-2} by R array and the process repeated, again using length- R DFT's. After M stages of

length- R DFT's with TF multiplications interleaved, the DFT is complete. The flow graph of a length-2 DFT is given in [Figure 1](#) and is called a butterfly because of its shape. The flow graph of the complete length-8 radix-2 FFT is shown in [Figure 2](#).



A Radix-2 Butterfly



Length-8 Radix-2 FFT Flow Graph

This flow-graph, the twiddle factor map of [\[link\]](#), and the basic equation [\[link\]](#) should be completely understood before going further.

A very efficient indexing scheme has evolved over the years that results in a compact and efficient computer program. A FORTRAN program is given below that implements the radix-2 FFT. It should be studied [\[link\]](#) to see how it implements [\[link\]](#) and the flow-graph representation.

```

N2 = N
DO 10 K = 1, M
  N1 = N2
  N2 = N2/2
  E = 6.28318/N1
  A = 0

```

```

DO 20 J = 1, N2
  C = COS (A)
  S = -SIN (A)
  A = J*E
  DO 30 I = J, N, N1
    L = I + N2
    XT = X(I) - X(L)
    X(I) = X(I) + X(L)
    YT = Y(I) - Y(L)
    Y(I) = Y(I) + Y(L)
    X(L) = XT*C - YT*S
    Y(L) = XT*S + YT*C
  30 CONTINUE
20 CONTINUE
10 CONTINUE

```

A Radix-2 Cooley-Tukey FFT Program

This discussion, the flow graph of [Winograd's Short DFT Algorithms: Figure 2](#) and the program of [\[link\]](#) are all based on the input index map of [Multidimensional Index Mapping: Equation 6](#) and [\[link\]](#) and the calculations are performed in-place. According to [Multidimensional Index Mapping: In-Place Calculation of the DFT and Scrambling](#), this means the output is scrambled in bit-reversed order and should be followed by an unscrambler to give the DFT in proper order. This formulation is called a decimation-in-frequency FFT [\[link\]](#), [\[link\]](#), [\[link\]](#). A very similar algorithm based on the output index map can be derived which is called a decimation-in-time FFT. Examples of FFT programs are found in [\[link\]](#) and in the Appendix of this book.

Modifications to the Basic Cooley-Tukey FFT

Soon after the paper by Cooley and Tukey, there were improvements and extensions made. One very important discovery was the improvement in efficiency by using a larger radix of 4, 8 or even 16. For example, just as for the radix-2 butterfly, there are no multiplications required for a length-4 DFT, and therefore, a radix-4 FFT would have only twiddle factor multiplications. Because there are half as many stages in a radix-4 FFT, there would be half as many multiplications as in a radix-2 FFT. In practice, because some of the multiplications are by unity, the improvement is not by a factor of two, but it is significant. A radix-4 FFT is easily developed from the basic radix-2 structure by replacing the length-2 butterfly by a length-4 butterfly and making a few other modifications. Programs can be found in [\[link\]](#) and operation counts will be given in ["Evaluation of the Cooley-Tukey FFT Algorithms"](#).

Increasing the radix to 8 gives some improvement but not as much as from 2 to 4. Increasing it to 16 is theoretically promising but the small decrease in multiplications is somewhat offset by an increase in additions and the program becomes rather long. Other radices are not attractive because they generally require a substantial number of multiplications and additions in the butterflies.

The second method of reducing arithmetic is to remove the unnecessary TF multiplications by plus or minus unity or by plus or minus the square root of minus one. This occurs when the exponent of W_N is zero or a multiple of $N/4$. A reduction of additions as well as multiplications is achieved by removing these extraneous complex multiplications since a complex multiplication requires at least two real additions. In a program, this reduction is usually achieved by having special butterflies for the cases where the TF is one or j . As many as four special butterflies may be necessary to remove all unnecessary arithmetic, but in many cases there will be no practical improvement above two or three.

In addition to removing multiplications by one or j , there can be a reduction in multiplications by using a special butterfly for TFs with $W_{N/8}$, which have equal real and imaginary parts. Also, for computers or hardware with multiplication considerably slower than addition, it is desirable to use an algorithm for complex multiplication that requires three multiplications and three additions rather than the conventional four multiplications and two additions. Note that this gives no reduction in the total number of arithmetic operations, but does give a trade of multiplications for additions. This is one reason not to use complex data types in programs but to explicitly program complex arithmetic.

A time-consuming and unnecessary part of the execution of a FFT program is the calculation of the sine and cosine terms which are the real and imaginary parts of the TFs. There are basically three approaches to obtaining the sine and cosine values. They can be calculated as needed which is what is done in the sample program above. One value per stage can be calculated and the others recursively calculated from those. That method is fast but suffers from accumulated round-off errors. The fastest method is to fetch precalculated values from a stored table. This has the disadvantage of requiring considerable memory space.

If all the N DFT values are not needed, special forms of the FFT can be developed using a process called pruning [\[link\]](#) which removes the operations concerned with the unneeded outputs.

Special algorithms are possible for cases with real data or with symmetric data [\[link\]](#). The decimation-in-time algorithm can be easily modified to transform real data and save half the arithmetic required for complex data [\[link\]](#). There are numerous other modifications to deal with special hardware considerations such as an array processor or a special microprocessor such as the Texas Instruments TMS320. Examples of programs that deal with some of these items can be found in [\[link\]](#), [\[link\]](#), [\[link\]](#).

The Split-Radix FFT Algorithm

Recently several papers [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#) have been published on algorithms to calculate a length- 2^M DFT more efficiently than a Cooley-Tukey FFT of any radix. They all have the same computational complexity and are optimal for lengths up through 16 and until recently was thought to give the best total add-multiply count possible for any power-of-two length. Yavne published an algorithm with the same computational complexity in 1968 [\[link\]](#), but it went largely unnoticed. Johnson and Frigo have recently reported the first improvement in almost 40 years [\[link\]](#). The reduction in total operations is only a few percent, but it is a reduction.

The basic idea behind the split-radix FFT (SRFFT) as derived by Duhamel and Hollmann [\[link\]](#), [\[link\]](#) is the application of a radix-2 index map to the even-indexed terms and a radix-4 map to the odd-indexed terms. The basic definition of the DFT

Equation:

$$C_k = \sum_{n=0}^{N-1} x_n W^{nk}$$

with $W = e^{-j2\pi/N}$ gives

Equation:

$$C_{2k} = \sum_{n=0}^{N/2-1} [x_n + x_{n+N/2}] W^{2nk}$$

for the even index terms, and

Equation:

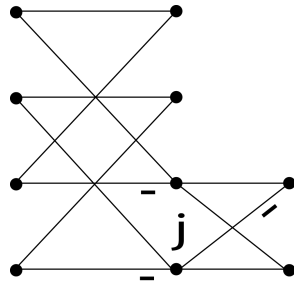
$$C_{4k+1} = \sum_{n=0}^{N/4-1} [(x_n - x_{n+N/2}) - j(x_{n+N/4} - x_{n+3N/4})] W^n W^{4nk}$$

and

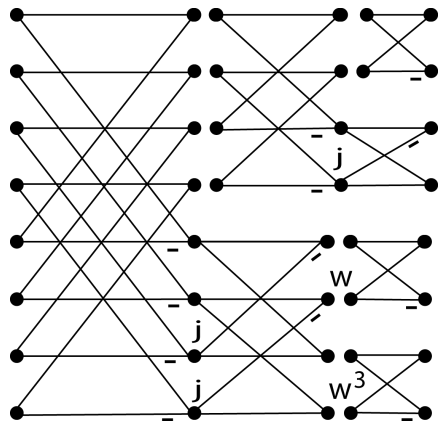
Equation:

$$C_{4k+3} = \sum_{n=0}^{N/4-1} [(x_n - x_{n+N/2}) + j(x_{n+N/4} - x_{n+3N/4})] W^{3n} W^{4nk}$$

for the odd index terms. This results in an L-shaped “butterfly” shown in [\[link\]](#) which relates a length-N DFT to one length-N/2 DFT and two length-N/4 DFT's with twiddle factors. Repeating this process for the half and quarter length DFT's until scalars result gives the SRFFT algorithm in much the same way the decimation-in-frequency radix-2 Cooley-Tukey FFT is derived [\[link\]](#), [\[link\]](#), [\[link\]](#). The resulting flow graph for the algorithm calculated in place looks like a radix-2 FFT except for the location of the twiddle factors. Indeed, it is the location of the twiddle factors that makes this algorithm use less arithmetic. The L-shaped SRFFT butterfly [\[link\]](#) advances the calculation of the top half by one of the M stages while the lower half, like a radix-4 butterfly, calculates two stages at once. This is illustrated for $N = 8$ in [\[link\]](#).



SRFFT Butterfly



Length-8 SRFFT

Unlike the fixed radix, mixed radix or variable radix Cooley-Tukey FFT or even the prime factor algorithm or Winograd Fourier transform algorithm, the Split-Radix FFT does not progress completely stage by stage, or, in terms of indices, does not complete each nested sum in order. This is perhaps better seen from the polynomial formulation of Martens [\[link\]](#). Because of this, the indexing is somewhat more complicated than the conventional Cooley-Tukey program.

A FORTRAN program is given below which implements the basic decimation-in-frequency split-radix FFT algorithm. The indexing scheme [\[link\]](#) of this program gives a structure very similar to the Cooley-Tukey programs in [\[link\]](#) and allows the same modifications and improvements such as decimation-in-time, multiple butterflies, table look-up of sine and cosine values, three real per complex multiply methods, and real data versions [\[link\]](#), [\[link\]](#).

```

SUBROUTINE FFT(X,Y,N,M)
  N2 = 2*N
  DO 10 K = 1, M-1
    N2 = N2/2
    N4 = N2/4
    E = 6.283185307179586/N2
    A = 0
    DO 20 J = 1, N4
      A3 = 3*A
      CC1 = COS(A)
      SS1 = SIN(A)
      CC3 = COS(A3)
      SS3 = SIN(A3)
      A = J*E
      IS = J
      ID = 2*N2
40    DO 30 I0 = IS, N-1, ID
      I1 = I0 + N4
      I2 = I1 + N4
      I3 = I2 + N4
      R1 = X(I0) - X(I2)
      X(I0) = X(I0) + X(I2)
      R2 = X(I1) - X(I3)

```

```

X(I1) = X(I1) + X(I3)
S1    = Y(I0) - Y(I2)
Y(I0) = Y(I0) + Y(I2)
S2    = Y(I1) - Y(I3)
Y(I1) = Y(I1) + Y(I3)
S3    = R1 - S2
R1    = R1 + S2
S2    = R2 - S1
R2    = R2 + S1
X(I2) = R1*CC1 - S2*SS1
Y(I2) = -S2*CC1 - R1*SS1
X(I3) = S3*CC3 + R2*SS3
Y(I3) = R2*CC3 - S3*SS3
30      CONTINUE
      IS = 2*ID - N2 + J
      ID = 4*ID
      IF (IS.LT.N) GOTO 40
20      CONTINUE
10      CONTINUE
      IS = 1
      ID = 4
50      DO 60 I0 = IS, N, ID
          I1 = I0 + 1
          R1 = X(I0)
          X(I0) = R1 + X(I1)
          X(I1) = R1 - X(I1)
          R1 = Y(I0)
          Y(I0) = R1 + Y(I1)
60      Y(I1) = R1 - Y(I1)
          IS = 2*ID - 1
          ID = 4*ID
      IF (IS.LT.N) GOTO 50

```

Split-Radix FFT FORTRAN Subroutine

As was done for the other decimation-in-frequency algorithms, the input index map is used and the calculations are done in place resulting in the output being in bit-reversed order. It is the three statements following label 30 that do the special indexing required by the SRFFT. The last stage is length-2 and, therefore, inappropriate for the standard L-shaped butterfly, so it is calculated separately in the DO 60 loop. This program is considered a one-butterfly version. A second butterfly can be added just before statement 40 to remove the unnecessary multiplications by unity. A third butterfly can be added to reduce the number of real multiplications from four to two for the complex multiplication when W has equal real and imaginary parts. It is also possible to reduce the arithmetic for the two-butterfly case and to reduce the data transfers by directly programming a length-4 and length-8 butterfly to replace the last three stages. This is called a two-butterfly-plus version. Operation counts for the one, two, two-plus and three butterfly SRFFT programs are given in the next section. Some details can be found in [\[link\]](#).

The special case of a SRFFT for real data and symmetric data is discussed in [\[link\]](#). An application of the decimation-in-time SRFFT to real data is given in [\[link\]](#). Application to convolution is made in [\[link\]](#), to the discrete Hartley transform in [\[link\]](#), [\[link\]](#), to calculating the discrete cosine transform in [\[link\]](#), and could be made to calculating number theoretic transforms.

An improvement in operation count has been reported by Johnson and Frigo [\[link\]](#) which involves a scaling of multiplying factors. The improvement is small but until this result, it was generally thought the Split-Radix FFT was optimal for total floating point operation count.

Evaluation of the Cooley-Tukey FFT Algorithms

The evaluation of any FFT algorithm starts with a count of the real (or floating point) arithmetic. [\[link\]](#) gives the number of real multiplications and additions required to calculate a length-N FFT of complex data. Results of programs with one, two, three and five butterflies are given to show the improvement that can be expected from removing unnecessary multiplications and additions. Results of radices two, four, eight and sixteen for the Cooley-Tukey FFT as well as of the split-radix FFT are given to show the relative merits of the various structures. Comparisons of these data should be made with the table of counts for the PFA and WFTA programs in [The Prime Factor and Winograd Fourier Transform Algorithms: Evaluation of the PFA and WFTA](#). All programs use the four-multiply-two-add complex multiply algorithm. A similar table can be developed for the three-multiply-three-add algorithm, but the relative results are the same.

From the table it is seen that a greater improvement is obtained going from radix-2 to 4 than from 4 to 8 or 16. This is partly because length 2 and 4 butterflies have no multiplications while length 8, 16 and higher do. It is also seen that going from one to two butterflies gives more improvement than going from two to higher values. From an operation count point of view and from practical experience, a three butterfly radix-4 or a two butterfly radix-8 FFT is a good compromise. The radix-8 and 16 programs become long, especially with multiple butterflies, and they give a limited choice of transform length unless combined with some length 2 and 4 butterflies.

N	M1	M2	M3	M5	A1	A2	A3	A5
2	4	0	0	0	6	4	4	4
4	16	4	0	0	24	18	16	16
8	48	20	8	4	72	58	52	52
16	128	68	40	28	192	162	148	148
32	320	196	136	108	480	418	388	388
64	768	516	392	332	1152	1026	964	964
128	1792	1284	1032	908	2688	2434	2308	2308
256	4096	3076	2568	2316	6144	5634	5380	5380
512	9216	7172	6152	5644	13824	12802	12292	12292
1024	20480	16388	14344	13324	30720	28674	27652	27652
2048	45056	36868	32776	30732	67584	63490	61444	61444

4096	98304	81924	73736	69644	147456	139266	135172	135172
4	12	0	0	0	22	16	16	16
16	96	36	28	24	176	146	144	144
64	576	324	284	264	1056	930	920	920
256	3072	2052	1884	1800	5632	5122	5080	5080
1024	15360	11268	10588	10248	28160	26114	25944	25944
4096	73728	57348	54620	53256	135168	126978	126296	126296
8	32	4	4	4	66	52	52	52
64	512	260	252	248	1056	930	928	928
512	6144	4100	4028	3992	12672	11650	11632	11632
4096	65536	49156	48572	48280	135168	126978	126832	126832
16	80	20	20	20	178	148	148	148
256	2560	1540	1532	1528	5696	5186	5184	5184
4096	61440	45060	44924	44856	136704	128514	128480	128480
2	0	0	0	0	4	4	4	4
4	8	0	0	0	20	16	16	16
8	24	8	4	4	60	52	52	52
16	72	32	28	24	164	144	144	144
32	184	104	92	84	412	372	372	372
64	456	288	268	248	996	912	912	912
128	1080	744	700	660	2332	2164	2164	2164
256	2504	1824	1740	1656	5348	5008	5008	5008
512	5688	4328	4156	3988	12060	11380	11380	11380
1024	12744	10016	9676	9336	26852	25488	25488	25488
2048	28216	22760	22076	21396	59164	56436	56436	56436
4096	61896	50976	49612	48248	129252	123792	123792	123792

Number of Real Multiplications and Additions for Complex Single Radix FFTs

NUMBER OF REAL MULTIPLICATIONS AND ADDITIONS FOR COMPLEX SINGLE RADIX FFTS

In [\[link\]](#) M_i and A_i refer to the number of real multiplications and real additions used by an FFT with i separately written butterflies. The first block has the counts for Radix-2, the second for Radix-4, the third for Radix-8, the fourth for Radix-16, and the last for the Split-Radix FFT. For the split-radix FFT, M_3 and A_3 refer to the two- butterfly-plus program and M_5 and A_5 refer to the three-butterfly program.

The first evaluations of FFT algorithms were in terms of the number of real multiplications required as that was the slowest operation on the computer and, therefore, controlled the execution speed. Later with hardware arithmetic both the number of multiplications and additions became important. Modern systems have arithmetic speeds such that indexing and data transfer times become important factors. Morris [\[link\]](#) has looked at some of these problems and has developed a procedure called autogen to write partially straight-line program code to significantly reduce overhead and speed up FFT run times. Some hardware, such as the TMS320 signal processing chip, has the multiply and add operations combined. Some machines have vector instructions or have parallel processors. Because the execution speed of an FFT depends not only on the algorithm, but also on the hardware architecture and compiler, experiments must be run on the system to be used.

In many cases the unscrambler or bit-reverse-counter requires 10% of the execution time, therefore, if possible, it should be eliminated. In high-speed convolution where the convolution is done by multiplication of DFT's, a decimation-in-frequency FFT can be combined with a decimation-in-time inverse FFT to require no unscrambler. It is also possible for a radix-2 FFT to do the unscrambling inside the FFT but the structure is not very regular [\[link\]](#), [\[link\]](#). Special structures can be found in [\[link\]](#) and programs for data that are real or have special symmetries are in [\[link\]](#), [\[link\]](#), [\[link\]](#).

Although there can be significant differences in the efficiencies of the various Cooley-Tukey and Split-Radix FFTs, the number of multiplications and additions for all of them is on the order of $N \log N$. That is fundamental to the class of algorithms.

The Quick Fourier Transform, An FFT based on Symmetries

The development of fast algorithms usually consists of using special properties of the algorithm of interest to remove redundant or unnecessary operations of a direct implementation. The discrete Fourier transform (DFT) defined by

Equation:

$$C(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}$$

where

Equation:

$$W_N = e^{-j2\pi/N}$$

has enormous capacity for improvement of its arithmetic efficiency. Most fast algorithms use the periodic and symmetric properties of its basis functions. The classical Cooley-Tukey FFT and prime factor FFT [\[link\]](#) exploit the periodic properties of the cosine and sine functions. Their use of the periodicities to share and, therefore, reduce arithmetic operations depends on the factorability of the length of the data to be transformed. For highly composite lengths, the number of floating-point operation is of order $N \log(N)$ and for prime lengths it is of order N^2 .

This section will look at an approach using the symmetric properties to remove redundancies. This possibility has long been recognized [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#) but has not been developed in any systematic way in the open literature. We will develop an algorithm, called the quick Fourier transform (QFT) [\[link\]](#), that will reduce the number of floating point operations necessary to compute the DFT by a factor of two to four over direct methods or Goertzel's method for prime lengths. Indeed, it seems the best general algorithm available for prime length DFTs. One can always do better by using Winograd type algorithms but they must be individually designed for each length. The Chirp Z-transform can be used for longer lengths.

Input and Output Symmetries

We use the fact that the cosine is an even function and the sine is an odd function. The kernel of the DFT or the basis functions of the expansion is given by

Equation:

$$W_N^{nk} = e^{-j2\pi nk/N} = \cos(2\pi nk/N) + j \sin(2\pi nk/N)$$

which has an even real part and odd imaginary part. If the data $x(n)$ are decomposed into their real and imaginary parts and those into their even and odd parts, we have

Equation:

$$x(n) = u(n) + jv(n) = [u_e(n) + u_o(n)] + j[v_e(n) + v_o(n)]$$

where the even part of the real part of $x(n)$ is given by

Equation:

$$u_e(n) = (u(n) + u(-n))/2$$

and the odd part of the real part is

Equation:

$$u_o(n) = (u(n) - u(-n))/2$$

with corresponding definitions of $v_e(n)$ and $v_o(n)$. Using [Convolution Algorithms: Equation 32](#) with a simpler notation, the DFT of [Convolution Algorithms: Equation 29](#) becomes

Equation:

$$C(k) = \sum_{n=0}^{N-1} (u + jv)(\cos - j \sin).$$

The sum over an integral number of periods of an odd function is zero and the sum of an even function over half of the period is one half the sum over the whole period. This causes [\[link\]](#) and [\[link\]](#) to become

Equation:

$$C(k) = \sum_{n=0}^{N/2-1} [u_e \cos + v_o \sin] + j[v_e \cos - v_o \sin].$$

for $k = 0, 1, 2, \dots, N - 1$.

The evaluation of the DFT using equation [\[link\]](#) requires half as many real multiplication and half as many real additions as evaluating it using [\[link\]](#) or [\[link\]](#). We have exploited the symmetries of the sine and cosine as functions of the time index n . This is independent of whether the length is composite or not. Another view of this formulation is that we have used the property of associativity of multiplication and addition. In other words, rather than multiply two data points by the same value of a sine or cosine then add the results, one should add the data points first then multiply the sum by the sine or cosine which requires one rather than two multiplications.

Next we take advantage of the symmetries of the sine and cosine as functions of the frequency index k . Using these symmetries on [\[link\]](#) gives

Equation:

$$C(k) = \sum_{n=0}^{N/2-1} [u_e \cos + v_o \sin] + j [v_e \cos - v_o \sin]$$

Equation:

$$C(N - k) = \sum_{n=0}^{N/2-1} [u_e \cos - v_o \sin] + j [v_e \cos + v_o \sin].$$

for $k = 0, 1, 2, \dots, N/2 - 1$. This again reduces the number of operations by a factor of two, this time because it calculates two output values at a time. The first reduction by a factor of two is always available. The second is possible only if both DFT values are needed. It is not available if you are calculating only one DFT value. The above development has not dealt with the details that arise with the difference between an even and an odd length. That is straightforward.

Further Reductions if the Length is Even

If the length of the sequence to be transformed is even, there are further symmetries that can be exploited. There will be four data values that are all multiplied by plus or minus the same sine or cosine value. This means a more complicated pre-addition process which is a generalization of the simple calculation of the even and odd parts in [\[link\]](#) and [\[link\]](#) will reduce the size of the order N^2 part of the algorithm by still another factor of two or four. If the length is divisible by 4, the process can be repeated. Indeed, if the length is a power of 2, one can show this process is equivalent to calculating the DFT in terms of discrete cosine and sine transforms [\[link\]](#), [\[link\]](#) with a resulting arithmetic complexity of order $N \log(N)$ and with a structure that is well suited to real data calculations and pruning.

If the flow-graph of the Cooley-Tukey FFT is compared to the flow-graph of the QFT, one notices both similarities and differences. Both progress in stages as the length is continually divided by two. The Cooley-Tukey algorithm uses the periodic properties of the sine and cosine to give the familiar horizontal tree of butterflies. The parallel diagonal lines in this graph represent the parallel stepping through the data in synchronism with the periodic basis functions. The QFT has diagonal lines that connect the first data point with the last, then the second with the next to last, and so on to give a "star" like picture. This is interesting in that one can look at the flow graph of an algorithm developed by some completely different strategy and often find sections with the parallel structures and other parts with the star structure. These must be using some underlying periodic and symmetric properties of the basis functions.

Arithmetic Complexity and Timings

A careful analysis of the QFT shows that $2N$ additions are necessary to compute the even and odd parts of the input data. This is followed by the length $N/2$ inner product that requires $4(N/2)^2 = N^2$ real multiplications and an equal number of additions. This is followed by the calculations necessary for the simultaneous calculations of the first half and last half of $C(k)$ which requires $4(N/2) = 2N$ real additions. This means the total QFT algorithm requires M^2 real multiplications and $N^2 + 4N$ real additions. These numbers along with those for the Goertzel algorithm [\[link\]](#), [\[link\]](#), [\[link\]](#) and the direct calculation of the DFT are included in the following table. Of the various order- N^2 DFT algorithms, the QFT seems to be the most efficient general method for an arbitrary length N .

Algorithm	Real Mults.	Real Adds	Trig Eval.
Direct DFT	$4 N^2$	$4 N^2$	$2 N^2$
Mod. 2nd Order Goertzel	$N^2 + N$	$2 N^2 + N$	N
QFT	N^2	$N^2 + 4N$	$2N$

Timings of the algorithms on a PC in milliseconds are given in the following table.

Algorithm	$N = 125$	$N = 256$
Direct DFT	4.90	19.83
Mod. 2O. Goertzel	1.32	5.55
QFT	1.09	4.50
Chirp + FFT	1.70	3.52

These timings track the floating point operation counts fairly well.

Conclusions

The QFT is a straight-forward DFT algorithm that uses all of the possible symmetries of the DFT basis function with no requirements on the length being composite. These ideas have been proposed before, but have not been published or clearly developed by [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#). It seems that the basic QFT is practical and useful as a general algorithm for lengths up to a hundred or so. Above that, the chirp z-transform [\[link\]](#) or other filter based methods will be superior. For special cases and shorter lengths, methods based on Winograd's theories will always be superior. Nevertheless, the QFT has a definite place in the array of DFT algorithms and is not well known. A Fortran program is included in the appendix.

It is possible, but unlikely, that further arithmetic reduction could be achieved using the fact that W_N has unity magnitude as was done in second-order Goertzel algorithm. It is also possible that some way of combining the Goertzel and QFT algorithm would have some advantages. A development of a complete QFT decomposition of a DFT of length- 2^M shows interesting structure [\[link\]](#), [\[link\]](#) and arithmetic complexity comparable to average Cooley-Tukey FFTs. It does seem better suited to real data calculations with pruning.

The Prime Factor and Winograd Fourier Transform Algorithms

The prime factor algorithm (PFA) and the Winograd Fourier transform algorithm (WFTA) are methods for efficiently calculating the DFT which use, and in fact, depend on the Type-1 index map from [Multidimensional Index Mapping: Equation 10](#) and [Multidimensional Index Mapping: Equation 6](#). The use of this index map preceded Cooley and Tukey's paper [\[link\]](#), [\[link\]](#) but its full potential was not realized until it was combined with Winograd's short DFT algorithms. The modern PFA was first presented in [\[link\]](#) and a program given in [\[link\]](#). The WFTA was first presented in [\[link\]](#) and programs given in [\[link\]](#), [\[link\]](#).

The number theoretic basis for the indexing in these algorithms may, at first, seem more complicated than in the Cooley-Tukey FFT; however, if approached from the general index mapping point of view of [Multidimensional Index Mapping](#), it is straightforward, and part of a common approach to breaking large problems into smaller ones. The development in this section will parallel that in [The Cooley-Tukey Fast Fourier Transform Algorithm](#).

The general index maps of [Multidimensional Index Mapping: Equation 6](#) and [Multidimensional Index Mapping: Equation 12](#) must satisfy the Type-1 conditions of [Multidimensional Index Mapping: Equation 7](#) and [Multidimensional Index Mapping: Equation 10](#) which are

Equation:

$$K_1 = aN_2 \quad \text{and} \quad K_2 = bN_1 \quad \text{with} \quad (K_1, N_1) = (K_2, N_2) = 1$$

Equation:

$$K_3 = cN_2 \quad \text{and} \quad K_4 = dN_1 \quad \text{with} \quad (K_3, N_1) = (K_4, N_2) = 1$$

The row and column calculations in [Multidimensional Index Mapping: Equation 15](#) are uncoupled by [Multidimensional Index Mapping: Equation 16](#) which for this case are

Equation:

$$((K_1 K_4))_N = ((K_2 K_3))_N = 0$$

In addition, to make each short sum a DFT, the K_i must also satisfy

Equation:

$$((K_1 K_3))_N = N_2 \quad \text{and} \quad ((K_2 K_4))_N = N_1$$

In order to have the smallest values for K_i , the constants in [\[link\]](#) are chosen to be
Equation:

$$a = b = 1, \quad c = ((N_2^{-1}))_N, \quad d = ((N_1^{-1}))_N$$

which gives for the index maps in [\[link\]](#)

Equation:

$$n = ((N_2 n_1 + N_1 n_2))_N$$

Equation:

$$k = ((K_3 k_1 + K_4 k_2))_N$$

The frequency index map is a form of the Chinese remainder theorem. Using these index maps, the DFT in [Multidimensional Index Mapping: Equation 15](#) becomes

Equation:

$$X = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_1-1} x \, W_{N_1}^{n_1 k_1} \, W_{N_2}^{n_2 k_2}$$

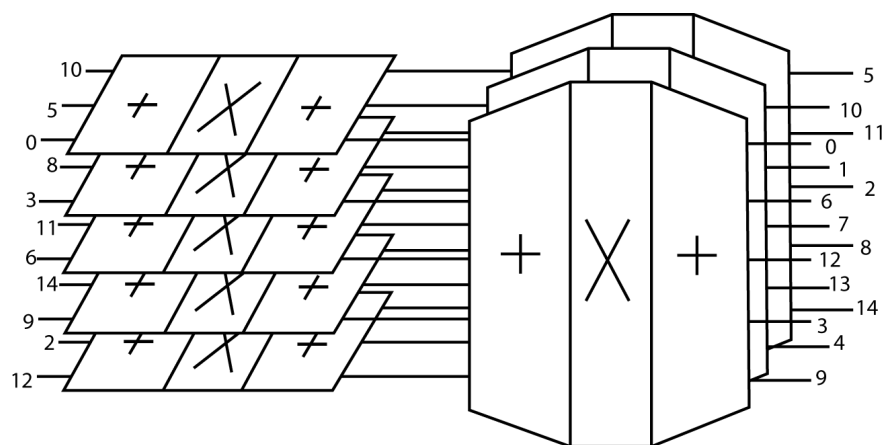
which is a pure two-dimensional DFT with no twiddle factors and the summations can be done in either order. Choices other than [\[link\]](#) could be used. For example, $a = b = c = d = 1$ will cause the input and output index map to be the same and, therefore, there will be no scrambling of the output order. The short summations in (96), however, will no longer be short DFT's [\[link\]](#).

An important feature of the short Winograd DFT's described in [Winograd's Short DFT Algorithms](#) that is useful for both the PFA and WFTA is the fact that the multiplier constants in [Winograd's Short DFT Algorithms: Equation 6](#) or [Winograd's Short DFT Algorithms: Equation 8](#) are either real or imaginary, never a general complex number. For that reason, multiplication by complex data requires only two real multiplications, not four. That is a very significant feature. It is also true that the j multiplier can be commuted from the D operator to the last part of the A^T operator. This means the D operator has only real multipliers and the calculations on real data remains real until the last stage. This can be seen by examining the short DFT modules in [\[link\]](#), [\[link\]](#) and in the appendices.

The Prime Factor Algorithm

If the DFT is calculated directly using [\[link\]](#), the algorithm is called a prime factor algorithm [\[link\]](#), [\[link\]](#) and was discussed in [Winograd's Short DFT Algorithms](#) and [Multidimensional Index Mapping: In-Place Calculation of the DFT and Scrambling](#). When the short DFT's are calculated by the very efficient algorithms of Winograd discussed in [Factoring the Signal Processing Operators](#), the PFA becomes a very powerful method that is as fast or faster than the best Cooley-Tukey FFT's [\[link\]](#), [\[link\]](#).

A flow graph is not as helpful with the PFA as it was with the Cooley-Tukey FFT, however, the following representation in [\[link\]](#) which combines Figures [Multidimensional Index Mapping: Figure 1](#) and [Winograd's Short DFT Algorithms: Figure 2](#) gives a good picture of the algorithm with the example of [Multidimensional Index Mapping: Equation 25](#)



A Prime Factor FFT for $N = 15$

If N is factored into three factors, the DFT of [\[link\]](#) would have three nested summations and would be a three-dimensional DFT. This principle extends to any number of factors; however, recall that the Type-1 map requires that all the factors be relatively prime. A very simple three-loop indexing scheme has been developed [\[link\]](#) which gives a compact, efficient PFA program for any number of factors. The basic program structure is illustrated in [\[link\]](#) with the short DFT's being omitted for clarity. Complete programs are given in [\[link\]](#) and in the appendices.

```
C-----PFA INDEXING LOOPS-----
      DO 10 K = 1, M
        N1 = NI(K)
```

```

        N2 = N/N1
        I(1) = 1
        DO 20 J = 1, N2
            DO 30 L=2, N1
                I(L) = I(L-1) + N2
                IF (I(L) .GT. N) I(L) = I(L) - N
30          CONTINUE
            GOTO (20,102,103,104,105), N1
            I(1) = I(1) + N1
20        CONTINUE
10      CONTINUE
        RETURN
C-----MODULE FOR N=2-----
102    R1      = X(I(1))
        X(I(1)) = R1 + X(I(2))
        X(I(2)) = R1 - X(I(2))
        R1      = Y(I(1))
        Y(I(1)) = R1 + Y(I(2))
        Y(I(2)) = R1 - Y(I(2))
        GOTO 20
C-----OTHER MODULES-----
103    Length-3 DFT
104    Length-4 DFT
105    Length-5 DFT
        etc.

```

Part of a FORTRAN PFA Program

As in the Cooley-Tukey program, the DO 10 loop steps through the M stages (factors of N) and the DO 20 loop calculates the N/N1 length-N1 DFT's. The input index map of [\[link\]](#) is implemented in the DO 30 loop and the statement just before label 20. In the PFA, each stage or factor requires a separately programmed module or butterfly. This lengthens the PFA program but an efficient Cooley-Tukey program will also require three or more butterflies.

Because the PFA is calculated in-place using the input index map, the output is scrambled. There are five approaches to dealing with this scrambled output. First, there are some applications where the output does not have to be unscrambled as in the case of high-speed convolution. Second, an unscrambler can be added after the PFA to give the output in correct order just as the bit-reversed-counter is used for the Cooley-Tukey FFT. A simple unscrambler is given in [\[link\]](#), [\[link\]](#) but it is not in place. The third method does the unscrambling in the modules while they are being calculated. This is

probably the fastest method but the program must be written for a specific length [\[link\]](#), [\[link\]](#). A fourth method is similar and achieves the unscrambling by choosing the multiplier constants in the modules properly [\[link\]](#). The fifth method uses a separate indexing method for the input and output of each module [\[link\]](#), [\[link\]](#).

The Winograd Fourier Transform Algorithm

The Winograd Fourier transform algorithm (WFTA) uses a very powerful property of the Type-1 index map and the DFT to give a further reduction of the number of multiplications in the PFA. Using an operator notation where F_1 represents taking row DFT's and F_2 represents column DFT's, the two-factor PFA of [\[link\]](#) is represented by **Equation:**

$$X = F_2 F_1 x$$

It has been shown [\[link\]](#), [\[link\]](#) that if each operator represents identical operations on each row or column, they commute. Since F_1 and F_2 represent length N_1 and N_2 DFT's, they commute and [\[link\]](#) can also be written

Equation:

$$X = F_1 F_2 x$$

If each short DFT in F is expressed by three operators as in [Winograd's Short DFT Algorithms: Equation 8](#) and [Winograd's Short DFT Algorithms: Figure 2](#), F can be factored as

Equation:

$$F = A^T D A$$

where A represents the set of additions done on each row or column that performs the residue reduction as [Winograd's Short DFT Algorithms: Equation 30](#). Because of the appearance of the flow graph of A and because it is the first operator on x , it is called a preweave operator [\[link\]](#). D is the set of M multiplications and A^T (or B^T or C^T) from [Winograd's Short DFT Algorithms: Equation 5](#) or [Winograd's Short DFT Algorithms: Equation 6](#) is the reconstruction operator called the postweave. Applying [\[link\]](#) to [\[link\]](#) gives

Equation:

$$X = A_2^T D_2 A_2 A_1^T D_1 A_1 x$$

This is the PFA of [\[link\]](#) and [\[link\]](#) where $A_1 D_1 A_1$ represents the row DFT's on the array formed from x . Because these operators commute, [\[link\]](#) can also be written as **Equation:**

$$X = A_2^T A_1^T D_2 D_1 A_2 A_1 x$$

or

Equation:

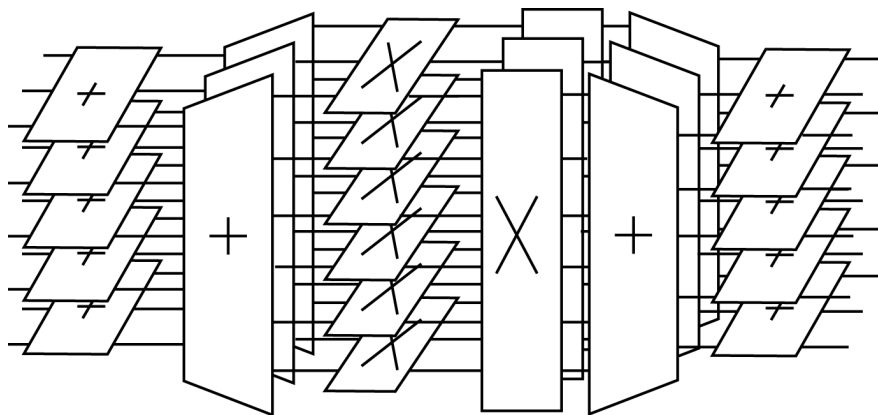
$$X = A_1^T A_2^T D_2 D_1 A_2 A_1 x$$

but the two adjacent multiplication operators can be premultiplied and the result represented by one operator $D = D_2 D_1$ which is no longer the same for each row or column. Equation [\[link\]](#) becomes

Equation:

$$X = A_1^T A_2^T D A_2 A_1 x$$

This is the basic idea of the Winograd Fourier transform algorithm. The commuting of the multiplication operators together in the center of the algorithm is called nesting and it results in a significant decrease in the number of multiplications that must be done at the execution of the algorithm. Pictorially, the PFA of [\[link\]](#) becomes [\[link\]](#) the WFTA in [\[link\]](#).



A Length-15 WFTA with Nested Multiplications

The rectangular structure of the preweave addition operators causes an expansion of the data in the center of the algorithm. The 15 data points in [\[link\]](#) become 18 intermediate values. This expansion is a major problem in programming the WFTA because it prevents a straightforward in-place calculation and causes an increase in the number of required additions and in the number of multiplier constants that must be precalculated and stored.

From [\[link\]](#) and the idea of premultiplying the individual multiplication operators, it can be seen why the multiplications by unity had to be considered in [Winograd's Short DFT Algorithms: Table 1](#). Even if a multiplier in D_1 is unity, it may not be in D_2D_1 . In [\[link\]](#) with factors of three and five, there appear to be 18 multiplications required because of the expansion of the length-5 preweave operator, A_2 , however, one of multipliers in each of the length three and five operators is unity, so one of the 18 multipliers in the product is unity. This gives 17 required multiplications - a rather impressive reduction from the $15^2 = 225$ multiplications required by direct calculation. This number of 17 complex multiplications will require only 34 real multiplications because, as mentioned earlier, the multiplier constants are purely real or imaginary while the 225 complex multiplications are general and therefore will require four times as many real multiplications.

The number of additions depends on the order of the pre- and postweave operators. For example in the length-15 WFTA in [\[link\]](#), if the length-5 had been done first and last, there would have been six row addition preweaves in the preweave operator rather than the five shown. It is difficult to illustrate the algorithm for three or more factors of N , but the ideas apply to any number of factors. Each length has an optimal ordering of the pre- and postweave operators that will minimize the number of additions.

A program for the WFTA is not as simple as for the FFT or PFA because of the very characteristic that reduces the number of multiplications, the nesting. A simple two-factor example program is given in [\[link\]](#) and a general program can be found in [\[link\]](#), [\[link\]](#). The same lengths are possible with the PFA and WFTA and the same short DFT modules can be used, however, the multiplies in the modules must occur in one place for use in the WFTA.

Modifications of the PFA and WFTA Type Algorithms

In the previous section it was seen how using the permutation property of the elementary operators in the PFA allowed the nesting of the multiplications to reduce their number. It was also seen that a proper ordering of the operators could minimize the number of additions. These ideas have been extended in formulating a more general algorithm optimizing problem. If the DFT operator F in [\[link\]](#) is expressed in a still more factored form obtained from [Winograd's Short DFT Algorithms: Equation](#)

[30](#), a greater variety of ordering can be optimized. For example if the A operators have two factors

Equation:

$$F_1 = A_1^T A_1'^T D_1 A_1' A_1$$

The DFT in [\[link\]](#) becomes

Equation:

$$X = A_2^T A_2'^T D_2 A_2' A_2 A_1^T A_1'^T D_1 A_1' A_1 x$$

The operator notation is very helpful in understanding the central ideas, but may hide some important facts. It has been shown [\[link\]](#), [\[link\]](#) that operators in different F_i commute with each other, but the order of the operators within an F_i cannot be changed. They represent the matrix multiplications in [Winograd's Short DFT Algorithms: Equation 30](#) or [Winograd's Short DFT Algorithms: Equation 8](#) which do not commute.

This formulation allows a very large set of possible orderings, in fact, the number is so large that some automatic technique must be used to find the "best". It is possible to set up a criterion of optimality that not only includes the number of multiplications but the number of additions as well. The effects of relative multiply-add times, data transfer times, CPU register and memory sizes, and other hardware characteristics can be included in the criterion. Dynamic programming can then be applied to derive an optimal algorithm for a particular application [\[link\]](#). This is a very interesting idea as there is no longer a single algorithm, but a class and an optimizing procedure. The challenge is to generate a broad enough class to result in a solution that is close to a global optimum and to have a practical scheme for finding the solution.

Results obtained applying the dynamic programming method to the design of fairly long DFT algorithms gave algorithms that had fewer multiplications and additions than either a pure PFA or WFTA [\[link\]](#). It seems that some nesting is desirable but not total nesting for four or more factors. There are also some interesting possibilities in mixing the Cooley-Tukey with this formulation. Unfortunately, the twiddle factors are not the same for all rows and columns, therefore, operations cannot commute past a twiddle factor operator. There are ways of breaking the total algorithm into horizontal paths and using different orderings along the different paths [\[link\]](#), [\[link\]](#). In a sense, this is what the split-radix FFT does with its twiddle factors when compared to a conventional Cooley-Tukey FFT.

There are other modifications of the basic structure of the Type-1 index map DFT algorithm. One is to use the same index structure and conversion of the short DFT's to convolution as the PFA but to use some other method for the high-speed convolution. Table look-up of partial products based on distributed arithmetic to eliminate all multiplications [\[link\]](#) looks promising for certain very specific applications, perhaps for specialized VLSI implementation. Another possibility is to calculate the short convolutions using number-theoretic transforms [\[link\]](#), [\[link\]](#), [\[link\]](#). This would also require special hardware. Direct calculation of short convolutions is faster on certain pipelined processor such as the TMS-320 microprocessor [\[link\]](#).

Evaluation of the PFA and WFTA

As for the Cooley-Tukey FFT's, the first evaluation of these algorithms will be on the number of multiplications and additions required. The number of multiplications to compute the PFA in [\[link\]](#) is given by [Multidimensional Index Mapping: Equation 3](#). Using the notation that $T(N)$ is the number of multiplications or additions necessary to calculate a length- N DFT, the total number for a four-factor PFA of length- N , where $N = N_1 N_2 N_3 N_4$ is

Equation:

$$T(N) = N_1 N_2 N_3 T(N_4) + N_2 N_3 N_4 T(N_1) + N_3 N_4 N_1 T(N_2) + N_4 N_1 N_2 T(N_3)$$

The count of multiplies and adds in [\[link\]](#) are calculated from (105) with the counts of the factors taken from [Winograd's Short DFT Algorithms: Table 1](#). The list of lengths are those possible with modules in the program of length 2, 3, 4, 5, 7, 8, 9 and 16 as is true for the PFA in [\[link\]](#), [\[link\]](#) and the WFTA in [\[link\]](#), [\[link\]](#). A maximum of four relatively prime lengths can be used from this group giving 59 different lengths over the range from 2 to 5040. The radix-2 or split-radix FFT allows 12 different lengths over the same range. If modules of length 11 and 13 from [\[link\]](#) are added, the maximum length becomes 720720 and the number of different lengths becomes 239. Adding modules for 17, 19 and 25 from [\[link\]](#) gives a maximum length of 1163962800 and a very large and dense number of possible lengths. The length of the code for the longer modules becomes excessive and should not be included unless needed.

The number of multiplications necessary for the WFTA is simply the product of those necessary for the required modules, including multiplications by unity. The total number may contain some unity multipliers but it is difficult to remove them in a practical program. [\[link\]](#) contains both the total number (MULTS) and the number with the unity multiplies removed (RMULTS).

Calculating the number of additions for the WFTA is more complicated than for the PFA because of the expansion of the data moving through the algorithm. For example

the number of additions, TA, for the length-15 example in [\[link\]](#) is given by **Equation:**

$$TA(N) = N_2TA(N_1) + TM_1TA(N_2)$$

where $N_1 = 3$, $N_2 = 5$, TM_1 = the number of multiplies for the length-3 module and hence the expansion factor. As mentioned earlier there is an optimum ordering to minimize additions. The ordering used to calculate [\[link\]](#) is the ordering used in [\[link\]](#), [\[link\]](#) which is optimal in most cases and close to optimal in the others.

Length	PFA	PFA	WFTA	WFTA	WFTA
N	Mults	Adds	Mults	RMults	Adds
10	20	88	24	20	88
12	16	96	24	16	96
14	32	172	36	32	172
15	50	162	36	34	162
18	40	204	44	40	208
20	40	216	48	40	216
21	76	300	54	52	300
24	44	252	48	36	252
28	64	400	72	64	400
30	100	384	72	68	384
35	150	598	108	106	666
36	80	480	88	80	488

40	100	532	96	84	532
42	152	684	108	104	684
45	190	726	132	130	804
48	124	636	108	92	660
56	156	940	144	132	940
60	200	888	144	136	888
63	284	1236	198	196	1394
70	300	1336	216	212	1472
72	196	1140	176	164	1156
80	260	1284	216	200	1352
84	304	1536	216	208	1536
90	380	1632	264	260	1788
105	590	2214	324	322	2418
112	396	2188	324	308	2332
120	460	2076	288	276	2076
126	568	2724	396	392	3040
140	600	2952	432	424	3224
144	500	2676	396	380	2880
168	692	3492	432	420	3492
180	760	3624	528	520	3936
210	1180	4848	648	644	5256

240	1100	4812	648	632	5136
252	1136	5952	792	784	6584
280	1340	6604	864	852	7148
315	2050	8322	1188	1186	10336
336	1636	7908	972	956	8508
360	1700	8148	1056	1044	8772
420	2360	10536	1296	1288	11352
504	2524	13164	1584	1572	14428
560	3100	14748	1944	1928	17168
630	4100	17904	2376	2372	21932
720	3940	18276	2376	2360	21132
840	5140	23172	2592	2580	24804
1008	5804	29100	3564	3548	34416
1260	8200	38328	4752	4744	46384
1680	11540	50964	5832	5816	59064
2520	17660	82956	9504	9492	99068
5040	39100	179772	21384	21368	232668

Number of Real Multiplications and Additions for Complex PFA and WFTA FFTs

from [\[link\]](#) we see that compared to the PFA or any of the Cooley-Tukey FFT's, the WFTA has significantly fewer multiplications. For the shorter lengths, the WFTA and the PFA have approximately the same number of additions; however for longer lengths, the PFA has fewer and the Cooley-Tukey FFT's always have the fewest. If the total arithmetic, the number of multiplications plus the number of additions, is compared, the split-radix FFT, PFA and WFTA all have about the same count. Special versions of the PFA and WFTA have been developed for real data [\[link\]](#), [\[link\]](#).

The size of the Cooley-Tukey program is the smallest, the PFA next and the WFTA largest. The PFA requires the smallest number of stored constants, the Cooley-Tukey or split-radix FFT next, and the WFTA requires the largest number. For a DFT of approximately 1000, the PFA stores 28 constants, the FFT 2048 and the WFTA 3564. Both the FFT and PFA can be calculated in-place and the WFTA cannot. The PFA can be calculated in-order without an unscrambler. The radix-2 FFT can also, but it requires additional indexing overhead [\[link\]](#). The indexing and data transfer overhead is greatest for the WFTA because the separate preweave and postweave sections each require their indexing and pass through the complete data. The shorter modules in the PFA and WFTA and the butterflies in the radix 2 and 4 FFT's are more efficient than the longer ones because intermediate calculations can be kept in cpu registers rather than general memory [\[link\]](#). However, the shorter modules and radices require more passes through the data for a given approximate length. A proper comparison will require actual programs to be compiled and run on a particular machine. There are many open questions about the relationship of algorithms and hardware architecture.

Implementing FFTs in Practice

Discussion of the considerations involved in high-performance FFT implementations, which center largely on memory access and other non-arithmetic concerns, as illustrated by a case study of the FFTW library.

by Steven G. Johnson (Department of Mathematics, Massachusetts Institute of Technology) and Matteo Frigo (Cilk Arts, Inc.)

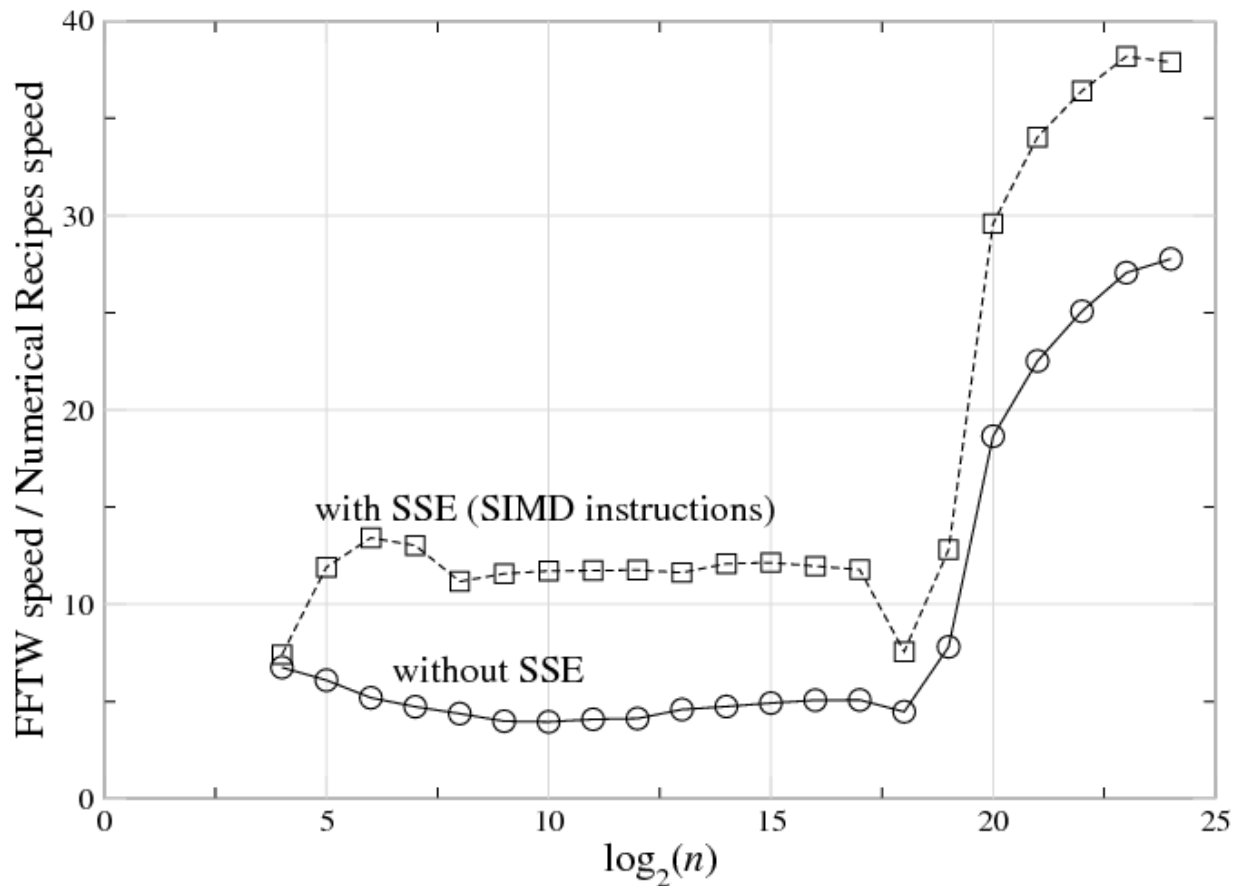
Introduction

Although there are a wide range of fast Fourier transform (FFT) algorithms, involving a wealth of mathematics from number theory to polynomial algebras, the vast majority of FFT implementations in practice employ some variation on the Cooley-Tukey algorithm [\[link\]](#). The Cooley-Tukey algorithm can be derived in two or three lines of elementary algebra. It can be implemented almost as easily, especially if only power-of-two sizes are desired; numerous popular textbooks list short FFT subroutines for power-of-two sizes, written in the language du jour. The implementation of the Cooley-Tukey algorithm, at least, would therefore seem to be a long-solved problem. In this chapter, however, we will argue that matters are not as straightforward as they might appear.

For many years, the primary route to improving upon the Cooley-Tukey FFT seemed to be reductions in the count of arithmetic operations, which often dominated the execution time prior to the ubiquity of fast floating-point hardware (at least on non-embedded processors). Therefore, great effort was expended towards finding new algorithms with reduced arithmetic counts [\[link\]](#), from Winograd's method to achieve $\Theta(n)$ multiplications [\[footnote\]](#) (at the cost of many more additions) [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#) to the split-radix variant on Cooley-Tukey that long achieved the lowest known total count of additions and multiplications for power-of-two sizes [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#) (but was recently improved upon [\[link\]](#), [\[link\]](#)). The question of the minimum possible arithmetic count continues to be of fundamental theoretical interest—it is not even known whether better than $\Theta(n \log n)$ complexity is possible, since $\Omega(n \log n)$ lower bounds on the count of additions have only been proven subject to restrictive assumptions about the algorithms [\[link\]](#), [\[link\]](#), [\[link\]](#).

Nevertheless, the difference in the number of arithmetic operations, for power-of-two sizes n , between the 1965 radix-2 Cooley-Tukey algorithm ($\sim 5n \log_2 n$ [\[link\]](#)) and the currently lowest-known arithmetic count ($\sim \frac{34}{9} n \log_2 n$ [\[link\]](#), [\[link\]](#)) remains only about 25%.

We employ the standard asymptotic notation of O for asymptotic upper bounds, Θ for asymptotic tight bounds, and Ω for asymptotic lower bounds [\[link\]](#).



The ratio of speed (1/time) between a highly optimized FFT (FFTW 3.1.2 [\[link\]](#), [\[link\]](#)) and a typical textbook radix-2 implementation (*Numerical Recipes in C* [\[link\]](#)) on a 3 GHz Intel Core Duo with the Intel C compiler 9.1.043, for single-precision complex-data DFTs of size n , plotted versus $\log_2 n$. Top line (squares) shows FFTW with SSE SIMD instructions enabled, which perform multiple arithmetic operations at once (see section); bottom line (circles) shows FFTW

with SSE disabled, which thus requires a similar number of arithmetic instructions to the textbook code. (This is not intended as a criticism of *Numerical Recipes*—simple radix-2 implementations are reasonable for pedagogy—but it illustrates the radical differences between straightforward and optimized implementations of FFT algorithms, even with similar arithmetic costs.) For $n \gtrsim 2^{19}$, the ratio increases because the textbook code becomes much slower (this happens when the DFT size exceeds the level-2 cache).

And yet there is a vast gap between this basic mathematical theory and the actual practice—highly optimized FFT packages are often an order of magnitude faster than the textbook subroutines, and the internal structure to achieve this performance is radically different from the typical textbook presentation of the “same” Cooley-Tukey algorithm. For example, [\[link\]](#) plots the ratio of benchmark speeds between a highly optimized FFT [\[link\]](#), [\[link\]](#) and a typical textbook radix-2 implementation [\[link\]](#), and the former is faster by a factor of 5–40 (with a larger ratio as n grows). Here, we will consider some of the reasons for this discrepancy, and some techniques that can be used to address the difficulties faced by a practical high-performance FFT implementation. [\[footnote\]](#)

We won't address the question of parallelization on multi-processor machines, which adds even greater difficulty to FFT implementation—although multi-processors are increasingly important, achieving good serial performance is a basic prerequisite for optimized parallel code, and is already hard enough!

In particular, in this chapter we will discuss some of the lessons learned and the strategies adopted in the FFTW library. FFTW [\[link\]](#), [\[link\]](#) is a widely used free-software library that computes the discrete Fourier transform (DFT) and its various special cases. Its performance is competitive even with manufacturer-optimized programs [\[link\]](#), and this performance is **portable** thanks the structure of the algorithms employed, self-optimization techniques, and highly optimized kernels (FFTW's **codelets**) generated by a special-purpose compiler.

This chapter is structured as follows. First ["Review of the Cooley-Tukey FFT"](#), we briefly review the basic ideas behind the Cooley-Tukey algorithm and define some common terminology, especially focusing on the many degrees of freedom that the abstract algorithm allows to implementations. Next, in ["Goals and Background of the FFTW Project"](#), we provide some context for FFTW's development and stress that performance, while it receives the most publicity, is not necessarily the most important consideration in the implementation of a library of this sort. Third, in ["FFTs and the Memory Hierarchy"](#), we consider a basic theoretical model of the computer memory hierarchy and its impact on FFT algorithm choices: quite general considerations push implementations towards large radices and explicitly recursive structure. Unfortunately, general considerations are not sufficient in themselves, so we will explain in ["Adaptive Composition of FFT Algorithms"](#) how FFTW self-optimizes for particular machines by selecting its algorithm at runtime from a composition of simple algorithmic steps. Furthermore, ["Generating Small FFT Kernels"](#) describes the utility and the principles of automatic code generation used to produce the highly optimized building blocks of this composition, FFTW's codelets. Finally, we will briefly consider an important non-performance issue, in ["Numerical Accuracy in FFTs"](#).

Review of the Cooley-Tukey FFT

The (forward, one-dimensional) discrete Fourier transform (DFT) of an array \mathbf{X} of n complex numbers is the array \mathbf{Y} given by

Equation:

$$\mathbf{Y}[k] = \sum_{\ell=0}^{n-1} \mathbf{X}[\ell] \omega_n^{\ell k},$$

where $0 \leq k < n$ and $\omega_n = \exp(-2\pi i/n)$ is a primitive root of unity. Implemented directly, [\[link\]](#) would require $\Theta(n^2)$ operations; fast Fourier transforms are $O(n \log n)$ algorithms to compute the same result. The most important FFT (and the one primarily used in FFTW) is known as the “Cooley-Tukey” algorithm, after the two authors who rediscovered and

popularized it in 1965 [\[link\]](#), although it had been previously known as early as 1805 by Gauss as well as by later re-inventors [\[link\]](#). The basic idea behind this FFT is that a DFT of a composite size $n = n_1 n_2$ can be re-expressed in terms of smaller DFTs of sizes n_1 and n_2 —essentially, as a two-dimensional DFT of size $n_1 \times n_2$ where the output is **transposed**. The choices of factorizations of n , combined with the many different ways to implement the data re-orderings of the transpositions, have led to numerous implementation strategies for the Cooley-Tukey FFT, with many variants distinguished by their own names [\[link\]](#), [\[link\]](#). FFTW implements a space of **many** such variants, as described in "[Adaptive Composition of FFT Algorithms](#)", but here we derive the basic algorithm, identify its key features, and outline some important historical variations and their relation to FFTW.

The Cooley-Tukey algorithm can be derived as follows. If n can be factored into $n = n_1 n_2$, [\[link\]](#) can be rewritten by letting $\ell = \ell_1 n_2 + \ell_2$ and $k = k_1 + k_2 n_1$. We then have:

Equation:

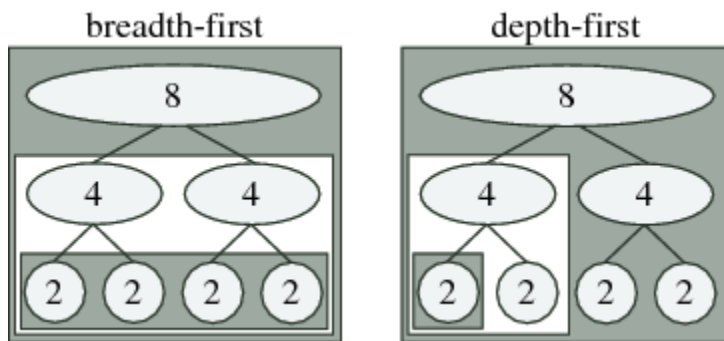
$$\mathbf{Y}[k_1 + k_2 n_1] = \sum_{\ell_2=0}^{n_2-1} \left[\left(\sum_{\ell_1=0}^{n_1-1} \mathbf{X}[\ell_1 n_2 + \ell_2] \omega_{n_1}^{\ell_1 k_1} \right) \omega_n^{\ell_2 k_1} \right] \omega_{n_2}^{\ell_2 k_2},$$

where $k_{1,2} = 0, \dots, n_{1,2} - 1$. Thus, the algorithm computes n_2 DFTs of size n_1 (the inner sum), multiplies the result by the so-called [\[link\]](#) **twiddle factors** $\omega_n^{\ell_2 k_1}$, and finally computes n_1 DFTs of size n_2 (the outer sum). This decomposition is then continued recursively. The literature uses the term **radix** to describe an n_1 or n_2 that is bounded (often constant); the small DFT of the radix is traditionally called a **butterfly**.

Many well-known variations are distinguished by the radix alone. A **decimation in time (DIT)** algorithm uses n_2 as the radix, while a **decimation in frequency (DIF)** algorithm uses n_1 as the radix. If multiple radices are used, e.g. for n composite but not a prime power, the algorithm is called **mixed radix**. A peculiar blending of radix 2 and 4 is called **split radix**, which was proposed to minimize the count of arithmetic operations [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#) although it has been superseded in this

regard [\[link\]](#), [\[link\]](#). FFTW implements both DIT and DIF, is mixed-radix with radices that are **adapted** to the hardware, and often uses much larger radices (e.g. radix 32) than were once common. On the other end of the scale, a “radix” of roughly \sqrt{n} has been called a **four-step** FFT algorithm (or **six-step**, depending on how many transposes one performs) [\[link\]](#); see ["FFTs and the Memory Hierarchy"](#) for some theoretical and practical discussion of this algorithm.

A key difficulty in implementing the Cooley-Tukey FFT is that the n_1 dimension corresponds to discontinuous inputs ℓ_1 in \mathbf{X} but contiguous outputs k_1 in \mathbf{Y} , and vice-versa for n_2 . This is a matrix transpose for a single decomposition stage, and the composition of all such transpositions is a (mixed-base) digit-reversal permutation (or **bit-reversal**, for radix 2). The resulting necessity of discontinuous memory access and data re-ordering hinders efficient use of hierarchical memory architectures (e.g., caches), so that the optimal execution order of an FFT for given hardware is non-obvious, and various approaches have been proposed.



Schematic of traditional breadth-first (left) vs. recursive depth-first (right) ordering for radix-2 FFT of size 8: the computations for each nested box are completed before doing anything else in the surrounding box. Breadth-first computation performs all butterflies of a given size at once, while depth-first computation completes one

subtransform entirely before moving on to the next (as in the algorithm below).

One ordering distinction is between recursion and iteration. As expressed above, the Cooley-Tukey algorithm could be thought of as defining a tree of smaller and smaller DFTs, as depicted in [\[link\]](#); for example, a textbook radix-2 algorithm would divide size n into two transforms of size $n/2$, which are divided into four transforms of size $n/4$, and so on until a base case is reached (in principle, size 1). This might naturally suggest a recursive implementation in which the tree is traversed “depth-first” as in [\[link\]](#)(right) and the algorithm of [\[link\]](#)—one size $n/2$ transform is solved completely before processing the other one, and so on. However, most traditional FFT implementations are non-recursive (with rare exceptions [\[link\]](#)) and traverse the tree “breadth-first” [\[link\]](#) as in [\[link\]](#)(left)—in the radix-2 example, they would perform n (trivial) size-1 transforms, then $n/2$ combinations into size-2 transforms, then $n/4$ combinations into size-4 transforms, and so on, thus making $\log_2 n$ passes over the whole array. In contrast, as we discuss in ["Discussion"](#), FFTW employs an explicitly recursive strategy that encompasses **both** depth-first and breadth-first styles, favoring the former since it has some theoretical and practical advantages as discussed in ["FFTs and the Memory Hierarchy"](#).

```

 $\mathbf{Y}[0, \dots, n - 1] \leftarrow \text{recfft2}(n, \mathbf{X}, \iota):$ 
IF  $n=1$  THEN
     $Y[0] \leftarrow X[0]$ 
ELSE
     $\mathbf{Y}[0, \dots, n/2 - 1] \leftarrow \text{recfft2}(n/2, \mathbf{X}, 2\iota)$ 
     $\mathbf{Y}[n/2, \dots, n - 1] \leftarrow \text{recfft2}(n/2, \mathbf{X} + \iota, 2\iota)$ 
    FOR  $k_1 = 0$  TO  $(n/2) - 1$  DO
         $t \leftarrow \mathbf{Y}[k_1]$ 
         $\mathbf{Y}[k_1] \leftarrow t + \omega_n^{k_1} \mathbf{Y}[k_1 + n/2]$ 
         $\mathbf{Y}[k_1 + n/2] \leftarrow t - \omega_n^{k_1} \mathbf{Y}[k_1 + n/2]$ 
    END FOR
END IF
depth-first recursive radix-2 DIT Cooley-
```

Tukey FFT to compute a DFT of a power-of-two size $n = 2^m$. The input is an array \mathbf{X} of length n with stride ι (i.e., the inputs are $\mathbf{X}[\ell\iota]$ for $\ell = 0, \dots, n - 1$) and the output is an array \mathbf{Y} of length n (with stride 1), containing the DFT of \mathbf{X} [Equation 1]. $\mathbf{X} + \iota$ denotes the array beginning with $\mathbf{X}[\iota]$. This algorithm operates out-of-place, produces in-order output, and does not require a separate bit-reversal stage.

A second ordering distinction lies in how the digit-reversal is performed. The classic approach is a single, separate digit-reversal pass following or preceding the arithmetic computations; this approach is so common and so deeply embedded into FFT lore that many practitioners find it difficult to imagine an FFT without an explicit bit-reversal stage. Although this pass requires only $O(n)$ time [\[link\]](#), it can still be non-negligible, especially if the data is out-of-cache; moreover, it neglects the possibility that data reordering during the transform may improve memory locality. Perhaps the oldest alternative is the Stockham **auto-sort** FFT [\[link\]](#), [\[link\]](#), which transforms back and forth between two arrays with each butterfly, transposing one digit each time, and was popular to improve contiguity of access for vector computers [\[link\]](#). Alternatively, an explicitly recursive style, as in FFTW, performs the digit-reversal implicitly at the “leaves” of its computation when operating out-of-place (see section ["Discussion"](#)). A simple example of this style, which computes in-order output using an out-of-place radix-2 FFT without explicit bit-reversal, is shown in the algorithm of [\[link\]](#) [corresponding to [\[link\]](#)(right)]. To operate in-place with $O(1)$ scratch storage, one can interleave small matrix transpositions with the butterflies [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), and a related strategy in FFTW [\[link\]](#) is briefly described by ["Discussion"](#).

Finally, we should mention that there are many FFTs entirely distinct from Cooley-Tukey. Three notable such algorithms are the **prime-factor algorithm** for $\gcd(n_1, n_2) = 1$ [\[link\]](#), along with Rader's [\[link\]](#) and Bluestein's [\[link\]](#), [\[link\]](#), [\[link\]](#) algorithms for prime n . FFTW implements the first two in its codelet generator for hard-coded n "[Generating Small FFT Kernels](#)" and the latter two for general prime n (sections "[Plans for prime sizes](#)" and "[Goals and Background of the FFTW Project](#)"). There is also the Winograd FFT [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), which minimizes the number of multiplications at the expense of a large number of additions; this trade-off is not beneficial on current processors that have specialized hardware multipliers.

Goals and Background of the FFTW Project

The FFTW project, begun in 1997 as a side project of the authors Frigo and Johnson as graduate students at MIT, has gone through several major revisions, and as of 2008 consists of more than 40,000 lines of code. It is difficult to measure the popularity of a free-software package, but (as of 2008) FFTW has been cited in over 500 academic papers, is used in hundreds of shipping free and proprietary software packages, and the authors have received over 10,000 emails from users of the software. Most of this chapter focuses on performance of FFT implementations, but FFTW would probably not be where it is today if that were the only consideration in its design. One of the key factors in FFTW's success seems to have been its flexibility in addition to its performance. In fact, FFTW is probably the most flexible DFT library available:

- FFTW is written in portable C and runs well on many architectures and operating systems.
- FFTW computes DFTs in $O(n \log n)$ time for any length n . (Most other DFT implementations are either restricted to a subset of sizes or they become $\Theta(n^2)$ for certain values of n , for example when n is prime.)
- FFTW imposes no restrictions on the rank (dimensionality) of multi-dimensional transforms. (Most other implementations are limited to one-dimensional, or at most two- and three-dimensional data.)

- FFTW supports multiple and/or strided DFTs; for example, to transform a 3-component vector field or a portion of a multi-dimensional array. (Most implementations support only a single DFT of contiguous data.)
- FFTW supports DFTs of real data, as well as of real symmetric/anti-symmetric data (also called discrete cosine/sine transforms).

Our design philosophy has been to first define the most general reasonable functionality, and then to obtain the highest possible performance without sacrificing this generality. In this section, we offer a few thoughts about why such flexibility has proved important, and how it came about that FFTW was designed in this way.

FFTW's generality is partly a consequence of the fact the FFTW project was started in response to the needs of a real application for one of the authors (a spectral solver for Maxwell's equations [\[link\]](#)), which from the beginning had to run on heterogeneous hardware. Our initial application required multi-dimensional DFTs of three-component vector fields (magnetic fields in electromagnetism), and so right away this meant: (i) multi-dimensional FFTs; (ii) user-accessible loops of FFTs of discontinuous data; (iii) efficient support for non-power-of-two sizes (the factor of eight difference between $n \times n \times n$ and $2n \times 2n \times 2n$ was too much to tolerate); and (iv) saving a factor of two for the common real-input case was desirable. That is, the initial requirements already encompassed most of the features above, and nothing about this application is particularly unusual.

Even for one-dimensional DFTs, there is a common misperception that one should always choose power-of-two sizes if one cares about efficiency. Thanks to FFTW's code generator (described in ["Generating Small FFT Kernels"](#)), we could afford to devote equal optimization effort to any n with small factors (2, 3, 5, and 7 are good), instead of mostly optimizing powers of two like many high-performance FFTs. As a result, to pick a typical example on the 3 GHz Core Duo processor of [\[link\]](#), $n = 3600 = 2^4 \cdot 3^2 \cdot 5^2$ and $n = 3840 = 2^8 \cdot 3 \cdot 5$ both execute faster than $n = 4096 = 2^{12}$. (And if there are factors one particularly cares about, one can generate code for them too.)

One initially missing feature was efficient support for large prime sizes; the conventional wisdom was that large-prime algorithms were mainly of academic interest, since in real applications (including ours) one has enough freedom to choose a highly composite transform size. However, the prime-size algorithms are fascinating, so we implemented Rader's $O(n \log n)$ prime- n algorithm [\[link\]](#) purely for fun, including it in FFTW 2.0 (released in 1998) as a bonus feature. The response was astonishingly positive—even though users are (probably) never **forced** by their application to compute a prime-size DFT, it is rather inconvenient to always worry that collecting an unlucky number of data points will slow down one's analysis by a factor of a million. The prime-size algorithms are certainly slower than algorithms for nearby composite sizes, but in interactive data-analysis situations the difference between 1 ms and 10 ms means little, while educating users to avoid large prime factors is hard.

Another form of flexibility that deserves comment has to do with a purely technical aspect of computer software. FFTW's implementation involves some unusual language choices internally (the FFT-kernel generator, described in ["Generating Small FFT Kernels"](#), is written in Objective Caml, a functional language especially suited for compiler-like programs), but its user-callable interface is purely in C with lowest-common-denominator datatypes (arrays of floating-point values). The advantage of this is that FFTW can be (and has been) called from almost any other programming language, from Java to Perl to Fortran 77. Similar lowest-common-denominator interfaces are apparent in many other popular numerical libraries, such as LAPACK [\[link\]](#). Language preferences arouse strong feelings, but this technical constraint means that modern programming dialects are best hidden from view for a numerical library.

Ultimately, very few scientific-computing applications should have performance as their top priority. Flexibility is often far more important, because one wants to be limited only by one's imagination, rather than by one's software, in the kinds of problems that can be studied.

FFTs and the Memory Hierarchy

There are many complexities of computer architectures that impact the optimization of FFT implementations, but one of the most pervasive is the memory hierarchy. On any modern general-purpose computer, memory is arranged into a hierarchy of storage devices with increasing size and decreasing speed: the fastest and smallest memory being the CPU registers, then two or three levels of cache, then the main-memory RAM, then external storage such as hard disks.[\[footnote\]](#) Most of these levels are managed automatically by the hardware to hold the most-recently-used data from the next level in the hierarchy.[\[footnote\]](#) There are many complications, however, such as limited cache associativity (which means that certain locations in memory cannot be cached simultaneously) and cache lines (which optimize the cache for contiguous memory access), which are reviewed in numerous textbooks on computer architectures. In this section, we focus on the simplest abstract principles of memory hierarchies in order to grasp their fundamental impact on FFTs.

A hard disk is utilized by “out-of-core” FFT algorithms for very large n [\[link\]](#), but these algorithms appear to have been largely superseded in practice by both the gigabytes of memory now common on personal computers and, for extremely large n , by algorithms for distributed-memory parallel computers.

This includes the registers: on current “x86” processors, the user-visible instruction set (with a small number of floating-point registers) is internally translated at runtime to RISC-like “ μ -ops” with a much larger number of physical **rename registers** that are allocated automatically.

Because access to memory is in many cases the slowest part of the computer, especially compared to arithmetic, one wishes to load as much data as possible in to the faster levels of the hierarchy, and then perform as much computation as possible before going back to the slower memory devices. This is called **temporal locality**: if a given datum is used more than once, we arrange the computation so that these usages occur as close together as possible in time.

Understanding FFTs with an ideal cache

To understand temporal-locality strategies at a basic level, in this section we will employ an idealized model of a cache in a two-level memory hierarchy, as defined in [\[link\]](#). This **ideal cache** stores Z data items from main memory (e.g. complex numbers for our purposes): when the processor loads a datum from memory, the access is quick if the datum is already in the cache (a **cache hit**) and slow otherwise (a **cache miss**, which requires the datum to be fetched into the cache). When a datum is loaded into the cache, [\[footnote\]](#) it must replace some other datum, and the ideal-cache model assumes that the optimal replacement strategy is used [\[link\]](#): the new datum replaces the datum that will not be needed for the longest time in the future; in practice, this can be simulated to within a factor of two by replacing the least-recently used datum [\[link\]](#), but ideal replacement is much simpler to analyze. Armed with this ideal-cache model, we can now understand some basic features of FFT implementations that remain essentially true even on real cache architectures. In particular, we want to know the **cache complexity**, the number $Q(n; Z)$ of cache misses for an FFT of size n with an ideal cache of size Z , and what algorithm choices reduce this complexity.

More generally, one can assume that a **cache line** of L consecutive data items are loaded into the cache at once, in order to exploit spatial locality. The ideal-cache model in this case requires that the cache be **tall**: $Z = \Omega(L^2)$ [\[link\]](#).

First, consider a textbook radix-2 algorithm, which divides n by 2 at each stage and operates breadth-first as in [\[link\]](#)(left), performing all butterflies of a given size at a time. If $n > Z$, then each pass over the array incurs $\Theta(n)$ cache misses to reload the data, and there are $\log_2 n$ passes, for $\Theta(n \log_2 n)$ cache misses in total—no temporal locality at all is exploited!

One traditional solution to this problem is **blocking**: the computation is divided into maximal blocks that fit into the cache, and the computations for each block are completed before moving on to the next block. Here, a block of Z numbers can fit into the cache [\[footnote\]](#) (not including storage for twiddle factors and so on), and thus the natural unit of computation is a sub-FFT of size Z . Since each of these blocks involves $\Theta(Z \log Z)$ arithmetic operations, and there are $\Theta(n \log n)$ operations overall, there must be $\Theta\left(\frac{n}{Z} \log_Z n\right)$ such blocks. More explicitly, one could use a radix-

Z Cooley-Tukey algorithm, breaking n down by factors of Z [or $\Theta(Z)$] until a size Z is reached: each stage requires n/Z blocks, and there are $\log_Z n$ stages, again giving $\Theta\left(\frac{n}{Z} \log_Z n\right)$ blocks overall. Since each block requires Z cache misses to load it into cache, the cache complexity Q_b of such a blocked algorithm is

Of course, $O(n)$ additional storage may be required for twiddle factors, the output data (if the FFT is not in-place), and so on, but these only affect the n that fits into cache by a constant factor and hence do not impact cache-complexity analysis. We won't worry about such constant factors in this section.

Equation:

$$Q_b(n; Z) = \Theta(n \log_Z n).$$

In fact, this complexity is rigorously **optimal** for Cooley-Tukey FFT algorithms [\[link\]](#), and immediately points us towards **large radices** (not radix 2!) to exploit caches effectively in FFTs.

However, there is one shortcoming of any blocked FFT algorithm: it is **cache aware**, meaning that the implementation depends explicitly on the cache size Z . The implementation must be modified (e.g. changing the radix) to adapt to different machines as the cache size changes. Worse, as mentioned above, actual machines have multiple levels of cache, and to exploit these one must perform multiple levels of blocking, each parameterized by the corresponding cache size. In the above example, if there were a smaller and faster cache of size $z < Z$, the size- Z sub-FFTs should themselves be performed via radix- z Cooley-Tukey using blocks of size z . And so on. There are two paths out of these difficulties: one is self-optimization, where the implementation automatically adapts itself to the hardware (implicitly including any cache sizes), as described in "[Adaptive Composition of FFT Algorithms](#)"; the other is to exploit **cache-oblivious** algorithms. FFTW employs both of these techniques.

The goal of cache-obliviousness is to structure the algorithm so that it exploits the cache without having the cache size as a parameter: the same code achieves the same asymptotic cache complexity regardless of the

cache size Z . An **optimal cache-oblivious** algorithm achieves the **optimal** cache complexity (that is, in an asymptotic sense, ignoring constant factors). Remarkably, optimal cache-oblivious algorithms exist for many problems, such as matrix multiplication, sorting, transposition, and FFTs [\[link\]](#). Not all cache-oblivious algorithms are optimal, of course—for example, the textbook radix-2 algorithm discussed above is “pessimal” cache-oblivious (its cache complexity is independent of Z because it always achieves the worst case!).

For instance, [\[link\]](#)(right) and the algorithm of [\[link\]](#) shows a way to obviously exploit the cache with a radix-2 Cooley-Tukey algorithm, by ordering the computation depth-first rather than breadth-first. That is, the DFT of size n is divided into two DFTs of size $n/2$, and one DFT of size $n/2$ is **completely finished** before doing **any** computations for the second DFT of size $n/2$. The two subtransforms are then combined using $n/2$ radix-2 butterflies, which requires a pass over the array and (hence n cache misses if $n > Z$). This process is repeated recursively until a base-case (e.g. size 2) is reached. The cache complexity $Q_2(n; Z)$ of this algorithm satisfies the recurrence

Equation:

$$Q_2(n; Z) = \begin{cases} n & n \leq Z \\ 2Q_2(n/2; Z) + \Theta(n) & \text{otherwise} \end{cases}.$$

The key property is this: once the recursion reaches a size $n \leq Z$, the subtransform fits into the cache and no further misses are incurred. The algorithm does not “know” this and continues subdividing the problem, of course, but all of those further subdivisions are in-cache because they are performed in the same depth-first branch of the tree. The solution of [\[link\]](#) is

Equation:

$$Q_2(n; Z) = \Theta(n \log \lceil n/Z \rceil).$$

This is worse than the theoretical optimum $Q_b(n; Z)$ from [\[link\]](#), but it is cache-oblivious (Z never entered the algorithm) and exploits at least **some**

temporal locality.[\[footnote\]](#) On the other hand, when it is combined with FFTW's self-optimization and larger radices in "[Adaptive Composition of FFT Algorithms](#)", this algorithm actually performs very well until n becomes extremely large. By itself, however, the algorithm of [\[link\]](#) must be modified to attain adequate performance for reasons that have nothing to do with the cache. These practical issues are discussed further in "[Cache-obliviousness in practice](#)".

This advantage of depth-first recursive implementation of the radix-2 FFT was pointed out many years ago by Singleton (where the “cache” was core memory) [\[link\]](#).

There exists a different recursive FFT that is **optimal** cache-oblivious, however, and that is the radix- \sqrt{n} “four-step” Cooley-Tukey algorithm (again executed recursively, depth-first) [\[link\]](#). The cache complexity Q_o of this algorithm satisfies the recurrence:

Equation:

$$Q_o(n; Z) = \begin{cases} n & n \leq Z \\ 2\sqrt{n}Q_o(\sqrt{n}; Z) + \Theta(n) & \text{otherwise} \end{cases}.$$

That is, at each stage one performs \sqrt{n} DFTs of size \sqrt{n} (recursively), then multiplies by the $\Theta(n)$ twiddle factors (and does a matrix transposition to obtain in-order output), then finally performs another \sqrt{n} DFTs of size \sqrt{n} . The solution of [\[link\]](#) is $Q_o(n; Z) = \Theta(n \log_Z n)$, the same as the optimal cache complexity [\[link\]](#)!

These algorithms illustrate the basic features of most optimal cache-oblivious algorithms: they employ a recursive divide-and-conquer strategy to subdivide the problem until it fits into cache, at which point the subdivision continues but no further cache misses are required. Moreover, a cache-oblivious algorithm exploits all levels of the cache in the same way, so an optimal cache-oblivious algorithm exploits a multi-level cache optimally as well as a two-level cache [\[link\]](#): the multi-level “blocking” is implicit in the recursion.

Cache-obliviousness in practice

Even though the radix- \sqrt{n} algorithm is optimal cache-oblivious, it does not follow that FFT implementation is a solved problem. The optimality is only in an asymptotic sense, ignoring constant factors, $O(n)$ terms, etcetera, all of which can matter a great deal in practice. For small or moderate n , quite different algorithms may be superior, as discussed in ["Memory strategies in FFTW"](#). Moreover, real caches are inferior to an ideal cache in several ways. The unsurprising consequence of all this is that cache-obliviousness, like any complexity-based algorithm property, does not absolve one from the ordinary process of software optimization. At best, it reduces the amount of memory/cache tuning that one needs to perform, structuring the implementation to make further optimization easier and more portable.

Perhaps most importantly, one needs to perform an optimization that has almost nothing to do with the caches: the recursion must be “coarsened” to amortize the function-call overhead and to enable compiler optimization. For example, the simple pedagogical code of the algorithm in [\[link\]](#) recurses all the way down to $n = 1$, and hence there are $\approx 2n$ function calls in total, so that every data point incurs a two-function-call overhead on average. Moreover, the compiler cannot fully exploit the large register sets and instruction-level parallelism of modern processors with an $n = 1$ function body.[\[footnote\]](#) These problems can be effectively erased, however, simply by making the base cases larger, e.g. the recursion could stop when $n = 32$ is reached, at which point a highly optimized hard-coded FFT of that size would be executed. In FFTW, we produced this sort of large base-case using a specialized code-generation program described in ["Generating Small FFT Kernels"](#).

In principle, it might be possible for a compiler to automatically coarsen the recursion, similar to how compilers can partially unroll loops. We are currently unaware of any general-purpose compiler that performs this optimization, however.

One might get the impression that there is a strict dichotomy that divides cache-aware and cache-oblivious algorithms, but the two are not mutually exclusive in practice. Given an implementation of a cache-oblivious strategy, one can further optimize it for the cache characteristics of a

particular machine in order to improve the constant factors. For example, one can tune the radices used, the transition point between the radix- \sqrt{n} algorithm and the bounded-radix algorithm, or other algorithmic choices as described in ["Memory strategies in FFTW"](#). The advantage of starting cache-aware tuning with a cache-oblivious approach is that the starting point already exploits all levels of the cache to some extent, and one has reason to hope that good performance on one machine will be more portable to other architectures than for a purely cache-aware “blocking” approach. In practice, we have found this combination to be very successful with FFTW.

Memory strategies in FFTW

The recursive cache-oblivious strategies described above form a useful starting point, but FFTW supplements them with a number of additional tricks, and also exploits cache-obliviousness in less-obvious forms.

We currently find that the general radix- \sqrt{n} algorithm is beneficial only when n becomes very large, on the order of $2^{20} \approx 10^6$. In practice, this means that we use at most a single step of radix- \sqrt{n} (two steps would only be used for $n \gtrsim 2^{40}$). The reason for this is that the implementation of radix \sqrt{n} is less efficient than for a bounded radix: the latter has the advantage that an entire radix butterfly can be performed in hard-coded loop-free code within local variables/registers, including the necessary permutations and twiddle factors.

Thus, for more moderate n , FFTW uses depth-first recursion with a bounded radix, similar in spirit to the algorithm of [\[link\]](#) but with much larger radices (radix 32 is common) and base cases (size 32 or 64 is common) as produced by the code generator of ["Generating Small FFT Kernels"](#). The self-optimization described in ["Adaptive Composition of FFT Algorithms"](#) allows the choice of radix and the transition to the radix- \sqrt{n} algorithm to be tuned in a cache-aware (but entirely automatic) fashion.

For small n (including the radix butterflies and the base cases of the recursion), hard-coded FFTs (FFTW's **codelets**) are employed. However,

this gives rise to an interesting problem: a codelet for (e.g.) $n = 64$ is ~ 2000 lines long, with hundreds of variables and over 1000 arithmetic operations that can be executed in many orders, so what order should be chosen? The key problem here is the efficient use of the CPU registers, which essentially form a nearly ideal, fully associative cache. Normally, one relies on the compiler for all code scheduling and register allocation, but but the compiler needs help with such long blocks of code (indeed, the general register-allocation problem is NP-complete). In particular, FFTW's generator knows more about the code than the compiler—the generator knows it is an FFT, and therefore it can use an optimal cache-oblivious schedule (analogous to the radix- \sqrt{n} algorithm) to order the code independent of the number of registers [\[link\]](#). The compiler is then used only for local “cache-aware” tuning (both for register allocation and the CPU pipeline).[\[footnote\]](#) As a practical matter, one consequence of this scheduler is that FFTW's machine-independent codelets are no slower than machine-specific codelets generated by an automated search and optimization over many possible codelet implementations, as performed by the SPIRAL project [\[link\]](#).

One practical difficulty is that some “optimizing” compilers will tend to greatly re-order the code, destroying FFTW's optimal schedule. With GNU gcc, we circumvent this problem by using compiler flags that explicitly disable certain stages of the optimizer.

(When implementing hard-coded base cases, there is another choice because a loop of small transforms is always required. Is it better to implement a hard-coded FFT of size 64, for example, or an unrolled loop of four size-16 FFTs, both of which operate on the same amount of data? The former should be more efficient because it performs more computations with the same amount of data, thanks to the $\log n$ factor in the FFT's $n \log n$ complexity.)

In addition, there are many other techniques that FFTW employs to supplement the basic recursive strategy, mainly to address the fact that cache implementations strongly favor accessing consecutive data—thanks to cache lines, limited associativity, and direct mapping using low-order address bits (accessing data at power-of-two intervals in memory, which is distressingly common in FFTs, is thus especially prone to cache-line

conflicts). Unfortunately, the known FFT algorithms inherently involve some non-consecutive access (whether mixed with the computation or in separate bit-reversal/transposition stages). There are many optimizations in FFTW to address this. For example, the data for several butterflies at a time can be copied to a small buffer before computing and then copied back, where the copies and computations involve more consecutive access than doing the computation directly in-place. Or, the input data for the subtransform can be copied from (discontiguous) input to (contiguous) output before performing the subtransform in-place (see ["Indirect plans"](#)), rather than performing the subtransform directly out-of-place (as in [algorithm 1](#)). Or, the order of loops can be interchanged in order to push the outermost loop from the first radix step [the ℓ_2 loop in [\[link\]](#)] down to the leaves, in order to make the input access more consecutive (see ["Discussion"](#)). Or, the twiddle factors can be computed using a smaller look-up table (fewer memory loads) at the cost of more arithmetic (see ["Numerical Accuracy in FFTs"](#)). The choice of whether to use any of these techniques, which come into play mainly for moderate n ($2^{13} < n < 2^{20}$), is made by the self-optimizing planner as described in the next section.

Adaptive Composition of FFT Algorithms

As alluded to several times already, FFTW implements a wide variety of FFT algorithms (mostly rearrangements of Cooley-Tukey) and selects the “best” algorithm for a given n automatically. In this section, we describe how such self-optimization is implemented, and especially how FFTW's algorithms are structured as a composition of algorithmic fragments. These techniques in FFTW are described in greater detail elsewhere [\[link\]](#), so here we will focus only on the essential ideas and the motivations behind them.

An FFT algorithm in FFTW is a composition of algorithmic steps called a **plan**. The algorithmic steps each solve a certain class of **problems** (either solving the problem directly or recursively breaking it into sub-problems of the same type). The choice of plan for a given problem is determined by a **planner** that selects a composition of steps, either by runtime measurements to pick the fastest algorithm, or by heuristics, or by loading a pre-computed plan. These three pieces: problems, algorithmic steps, and the planner, are discussed in the following subsections.

The problem to be solved

In early versions of FFTW, the only choice made by the planner was the sequence of radices [\[link\]](#), and so each step of the plan took a DFT of a given size n , possibly with discontinuous input/output, and reduced it (via a radix r) to DFTs of size n/r , which were solved recursively. That is, each step solved the following problem: given a size n , an **input pointer** \mathbf{I} , an **input stride** ι , an **output pointer** \mathbf{O} , and an **output stride** o , it computed the DFT of $\mathbf{I}[\ell\iota]$ for $0 \leq \ell < n$ and stored the result in $\mathbf{O}[ko]$ for $0 \leq k < n$. However, we soon found that we could not easily express many interesting algorithms within this framework; for example, **in-place** ($\mathbf{I} = \mathbf{O}$) FFTs that do not require a separate bit-reversal stage [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#). It became clear that the key issue was not the choice of algorithms, as we had first supposed, but the definition of the problem to be solved. Because only problems that can be expressed can be solved, the representation of a problem determines an outer bound to the space of plans that the planner can explore, and therefore it ultimately constrains FFTW's performance.

The difficulty with our initial $(n, \mathbf{I}, \iota, \mathbf{O}, o)$ problem definition was that it forced each algorithmic step to address only a single DFT. In fact, FFTs break down DFTs into **multiple** smaller DFTs, and it is the **combination** of these smaller transforms that is best addressed by many algorithmic choices, especially to rearrange the order of memory accesses between the subtransforms. Therefore, we redefined our notion of a problem in FFTW to be not a single DFT, but rather a **loop** of DFTs, and in fact **multiple nested loops** of DFTs. The following sections describe some of the new algorithmic steps that such a problem definition enables, but first we will define the problem more precisely.

DFT problems in FFTW are expressed in terms of structures called I/O tensors, [\[footnote\]](#) which in turn are described in terms of ancillary structures called I/O dimensions. An **I/O dimension** d is a triple $d = (n, \iota, o)$, where n is a non-negative integer called the **length**, ι is an integer called the **input stride**, and o is an integer called the **output stride**. An **I/O tensor** $t = \{d_1, d_2, \dots, d_\rho\}$ is a set of I/O dimensions. The non-negative integer $\rho = |t|$ is called the **rank** of the I/O tensor. A **DFT**

problem, denoted by $\text{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$, consists of two I/O tensors \mathbf{N} and \mathbf{V} , and of two **pointers** \mathbf{I} and \mathbf{O} . Informally, this describes $|\mathbf{V}|$ nested loops of $|\mathbf{N}|$ -dimensional DFTs with input data starting at memory location \mathbf{I} and output data starting at \mathbf{O} .

I/O tensors are unrelated to the tensor-product notation used by some other authors to describe FFT algorithms [\[link\]](#), [\[link\]](#).

For simplicity, let us consider only one-dimensional DFTs, so that $\mathbf{N} = \{(n, \iota, o)\}$ implies a *DFT* of length n on input data with stride ι and output data with stride o , much like in the original FFTW as described above. The main new feature is then the addition of zero or more “loops” \mathbf{V} . More formally, $\text{dft}(\mathbf{N}, \{(n, \iota, o)\} \cup \mathbf{V}, \mathbf{I}, \mathbf{O})$ is recursively defined as a “loop” of n problems: for all $0 \leq k < n$, do all computations in $\text{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I} + k \cdot \iota, \mathbf{O} + k \cdot o)$. The case of multi-dimensional DFTs is defined more precisely elsewhere [\[link\]](#), but essentially each I/O dimension in \mathbf{N} gives one dimension of the transform.

We call \mathbf{N} the **size** of the problem. The **rank** of a problem is defined to be the rank of its size (i.e., the dimensionality of the DFT). Similarly, we call \mathbf{V} the **vector size** of the problem, and the **vector rank** of a problem is correspondingly defined to be the rank of its vector size. Intuitively, the vector size can be interpreted as a set of “loops” wrapped around a single DFT, and we therefore refer to a single I/O dimension of \mathbf{V} as a **vector loop**. (Alternatively, one can view the problem as describing a DFT over a $|\mathbf{V}|$ -dimensional vector space.) The problem does not specify the order of execution of these loops, however, and therefore FFTW is free to choose the fastest or most convenient order.

DFT problem examples

A more detailed discussion of the space of problems in FFTW can be found in [\[link\]](#), but a simple understanding can be gained by examining a few examples demonstrating that the I/O tensor representation is sufficiently general to cover many situations that arise in practice, including some that are not usually considered to be instances of the DFT.

A single one-dimensional DFT of length n , with stride-1 input \mathbf{X} and output \mathbf{Y} , as in [\[link\]](#), is denoted by the problem $\text{dft}(\{(n, 1, 1)\}, \{\}, \mathbf{X}, \mathbf{Y})$ (no loops: vector-rank zero).

As a more complicated example, suppose we have an $n_1 \times n_2$ matrix \mathbf{X} stored as n_1 consecutive blocks of contiguous length- n_2 rows (this is called **row-major** format). The in-place DFT of all the **rows** of this matrix would be denoted by the problem $\text{dft}(\{(n_2, 1, 1)\}, \{(n_1, n_2, n_2)\}, \mathbf{X}, \mathbf{X})$: a length- n_1 loop of size- n_2 contiguous DFTs, where each iteration of the loop offsets its input/output data by a stride n_2 . Conversely, the in-place DFT of all the **columns** of this matrix would be denoted by $\text{dft}(\{(n_1, n_2, n_2)\}, \{(n_2, 1, 1)\}, \mathbf{X}, \mathbf{X})$: compared to the previous example, \mathbf{N} and \mathbf{V} are swapped. In the latter case, each DFT operates on discontinuous data, and FFTW might well choose to interchange the loops: instead of performing a loop of DFTs computed individually, the subtransforms themselves could act on n_2 -component vectors, as described in ["The space of plans in FFTW"](#).

A size-1 DFT is simply a copy $Y[0] = X[0]$, and here this can also be denoted by $\mathbf{N} = \{\}$ (rank zero, a “zero-dimensional” DFT). This allows FFTW's problems to represent many kinds of copies and permutations of the data within the same problem framework, which is convenient because these sorts of operations arise frequently in FFT algorithms. For example, to copy n consecutive numbers from \mathbf{I} to \mathbf{O} , one would use the rank-zero problem $\text{dft}(\{\}, \{(n, 1, 1)\}, \mathbf{I}, \mathbf{O})$. More interestingly, the in-place **transpose** of an $n_1 \times n_2$ matrix \mathbf{X} stored in row-major format, as described above, is denoted by $\text{dft}(\{\}, \{(n_1, n_2, 1), (n_2, 1, n_1)\}, \mathbf{X}, \mathbf{X})$ (rank zero, vector-rank two).

The space of plans in FFTW

Here, we describe a subset of the possible plans considered by FFTW; while not exhaustive [\[link\]](#), this subset is enough to illustrate the basic structure of FFTW and the necessity of including the vector loop(s) in the problem definition to enable several interesting algorithms. The plans that

we now describe usually perform some simple “atomic” operation, and it may not be apparent how these operations fit together to actually compute DFTs, or why certain operations are useful at all. We shall discuss those matters in ["Discussion"](#).

Roughly speaking, to solve a general DFT problem, one must perform three tasks. First, one must reduce a problem of arbitrary vector rank to a set of loops nested around a problem of vector rank 0, i.e., a single (possibly multi-dimensional) DFT. Second, one must reduce the multi-dimensional DFT to a sequence of of rank-1 problems, i.e., one-dimensional DFTs; for simplicity, however, we do not consider multi-dimensional DFTs below. Third, one must solve the rank-1, vector rank-0 problem by means of some DFT algorithm such as Cooley-Tukey. These three steps need not be executed in the stated order, however, and in fact, almost every permutation and interleaving of these three steps leads to a correct DFT plan. The choice of the set of plans explored by the planner is critical for the usability of the FFTW system: the set must be large enough to contain the fastest possible plans, but it must be small enough to keep the planning time acceptable.

Rank-0 plans

The rank-0 problem $\text{dft}(\{\}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ denotes a permutation of the input array into the output array. FFTW does not solve arbitrary rank-0 problems, only the following two special cases that arise in practice.

- When $|\mathbf{V}| = 1$ and $\mathbf{I} \neq \mathbf{O}$, FFTW produces a plan that copies the input array into the output array. Depending on the strides, the plan consists of a loop or, possibly, of a call to the ANSI C function `memcpy`, which is specialized to copy contiguous regions of memory.
- When $|\mathbf{V}| = 2$, $\mathbf{I} = \mathbf{O}$, and the strides denote a matrix-transposition problem, FFTW creates a plan that transposes the array in-place. FFTW implements the square transposition $\text{dft}(\{\}, \{(n, \iota, o), (n, o, \iota)\}, \mathbf{I}, \mathbf{O})$ by means of the cache-oblivious algorithm from [\[link\]](#), which is fast and, in theory, uses the cache optimally regardless of the cache size (using principles similar to those described in the section ["FFTs and the Memory Hierarchy"](#)). A

generalization of this idea is employed for non-square transpositions with a large common factor or a small difference between the dimensions, adapting algorithms from [\[link\]](#).

Rank-1 plans

Rank-1 DFT problems denote ordinary one-dimensional Fourier transforms. FFTW deals with most rank-1 problems as follows.

Direct plans

When the DFT rank-1 problem is “small enough” (usually, $n \leq 64$), FFTW produces a **direct plan** that solves the problem directly. These plans operate by calling a fragment of C code (a **codelet**) specialized to solve problems of one particular size, whose generation is described in ["Generating Small FFT Kernels"](#). More precisely, the codelets compute a loop ($|\mathbf{V}| \leq 1$) of small DFTs.

Cooley-Tukey plans

For problems of the form $\text{dft}(\{(n, \iota, o)\}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ where $n = rm$, FFTW generates a plan that implements a radix- r Cooley-Tukey algorithm ["Review of the Cooley-Tukey FFT"](#). Both decimation-in-time and decimation-in-frequency plans are supported, with both small fixed radices (usually, $r \leq 64$) produced by the codelet generator ["Generating Small FFT Kernels"](#) and also arbitrary radices (e.g. radix- \sqrt{n}).

The most common case is a **decimation in time (DIT)** plan, corresponding to a **radix** $r = n_2$ (and thus $m = n_1$) in the notation of ["Review of the Cooley-Tukey FFT"](#): it first solves $\text{dft}(\{(m, r \cdot \iota, o)\}, \mathbf{V} \cup \{(r, \iota, m \cdot o)\}, \mathbf{I}, \mathbf{O})$, then multiplies the output array \mathbf{O} by the twiddle factors, and finally solves $\text{dft}(\{(r, m \cdot o, m \cdot o)\}, \mathbf{V} \cup \{(m, o, o)\}, \mathbf{O}, \mathbf{O})$. For performance, the last two steps are not planned independently, but are fused together in a single

“twiddle” codelet—a fragment of C code that multiplies its input by the twiddle factors and performs a DFT of size r , operating in-place on \mathbf{O} .

Plans for higher vector ranks

These plans extract a vector loop to reduce a DFT problem to a problem of lower vector rank, which is then solved recursively. Any of the vector loops of \mathbf{V} could be extracted in this way, leading to a number of possible plans corresponding to different loop orderings.

Formally, to solve $\text{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$, where $\mathbf{V} = \{(n, \iota, o)\} \cup \mathbf{V}_1$, FFTW generates a loop that, for all k such that $0 \leq k < n$, invokes a plan for $\text{dft}(\mathbf{N}, \mathbf{V}_1, \mathbf{I} + k \cdot \iota, \mathbf{O} + k \cdot o)$.

Indirect plans

Indirect plans transform a DFT problem that requires some data shuffling (or discontinuous operation) into a problem that requires no shuffling plus a rank-0 problem that performs the shuffling.

Formally, to solve $\text{dft}(\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ where $|\mathbf{N}| > 0$, FFTW generates a plan that first solves $\text{dft}(\{\}, \mathbf{N} \cup \mathbf{V}, \mathbf{I}, \mathbf{O})$, and then solves $\text{dft}(\text{copy-o}(\mathbf{N}), \text{copy-o}(\mathbf{V}), \mathbf{O}, \mathbf{O})$. Here we define $\text{copy-o}(t)$ to be the I/O tensor $\{(n, o, o) \mid (n, \iota, o) \in t\}$: that is, it replaces the input strides with the output strides. Thus, an indirect plan first rearranges/copies the data to the output, then solves the problem in place.

Plans for prime sizes

As discussed in ["Goals and Background of the FFTW Project"](#), it turns out to be surprisingly useful to be able to handle large prime n (or large prime factors). **Rader plans** implement the algorithm from [\[link\]](#) to compute one-dimensional DFTs of prime size in $\Theta(n \log n)$ time. **Bluestein plans**

implement Bluestein's “chirp-z” algorithm, which can also handle prime n in $\Theta(n \log n)$ time [\[link\]](#), [\[link\]](#), [\[link\]](#). **Generic plans** implement a naive $\Theta(n^2)$ algorithm (useful for $n \lesssim 100$).

Discussion

Although it may not be immediately apparent, the combination of the recursive rules in ["The space of plans in FFTW"](#) can produce a number of useful algorithms. To illustrate these compositions, we discuss three particular issues: depth- vs. breadth-first, loop reordering, and in-place transforms.

size-30 DFT, depth-first:

$$\left\{ \begin{array}{l} \text{loop 3} \\ \left\{ \begin{array}{l} \text{size-5 direct codelet, vector size 2} \\ \text{size-2 twiddle codelet, vector size 5} \end{array} \right. \\ \text{size-3 twiddle codelet, vector size 10} \end{array} \right.$$

size-30 DFT, breadth-first:

$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{loop 3} \\ \text{size-5 direct codelet, vector size 2} \end{array} \right. \\ \left\{ \begin{array}{l} \text{loop 3} \\ \text{size-2 twiddle codelet, vector size 5} \end{array} \right. \\ \text{size-3 twiddle codelet, vector size 10} \end{array} \right.$$

Two possible decompositions for a size-30 DFT, both for the arbitrary choice of DIT radices 3 then 2 then 5, and prime-size codelets. Items grouped by a "{" result from the plan for a single sub-problem. In the depth-first case, the vector rank was reduced to zero as per ["Plans for](#)

[higher vector ranks](#)" before decomposing sub-problems, and vice-versa in the breadth-first case.

As discussed previously in sections ["Review of the Cooley-Tukey FFT"](#) and ["Understanding FFTs with an ideal cache"](#), the same Cooley-Tukey decomposition can be executed in either traditional breadth-first order or in recursive depth-first order, where the latter has some theoretical cache advantages. FFTW is explicitly recursive, and thus it can naturally employ a depth-first order. Because its sub-problems contain a vector loop that can be executed in a variety of orders, however, FFTW can also employ breadth-first traversal. In particular, a 1d algorithm resembling the traditional breadth-first Cooley-Tukey would result from applying ["Cooley-Tukey plans"](#) to completely factorize the problem size before applying the loop rule ["Plans for higher vector ranks"](#) to reduce the vector ranks, whereas depth-first traversal would result from applying the loop rule before factorizing each subtransform. These two possibilities are illustrated by an example in [\[link\]](#).

Another example of the effect of loop reordering is a style of plan that we sometimes call **vector recursion** (unrelated to “vector-radix” FFTs [\[link\]](#)). The basic idea is that, if one has a loop (vector-rank 1) of transforms, where the vector stride is smaller than the transform size, it is advantageous to push the loop towards the leaves of the transform decomposition, while otherwise maintaining recursive depth-first ordering, rather than looping “outside” the transform; i.e., apply the usual FFT to “vectors” rather than numbers. Limited forms of this idea have appeared for computing multiple FFTs on vector processors (where the loop in question maps directly to a hardware vector) [\[link\]](#). For example, Cooley-Tukey produces a unit **input**-stride vector loop at the top-level DIT decomposition, but with a large **output** stride; this difference in strides makes it non-obvious whether vector recursion is advantageous for the sub-problem, but for large transforms we often observe the planner to choose this possibility.

In-place 1d transforms (with no separate bit reversal pass) can be obtained as follows by a combination DIT and DIF plans ["Cooley-Tukey plans"](#) with transposes ["Rank-0 plans"](#). First, the transform is decomposed via a radix- p DIT plan into a vector of p transforms of size qm , then these are

decomposed in turn by a radix- q DIF plan into a vector (rank 2) of $p \times q$ transforms of size m . These transforms of size m have input and output at different places/strides in the original array, and so cannot be solved independently. Instead, an indirect plan "[Indirect plans](#)" is used to express the sub-problem as pq in-place transforms of size m , followed or preceded by an $m \times p \times q$ rank-0 transform. The latter sub-problem is easily seen to be m in-place $p \times q$ transposes (ideally square, i.e. $p = q$). Related strategies for in-place transforms based on small transposes were described in [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#); alternating DIT/DIF, without concern for in-place operation, was also considered in [\[link\]](#), [\[link\]](#).

The FFTW planner

Given a problem and a set of possible plans, the basic principle behind the FFTW planner is straightforward: construct a plan for each applicable algorithmic step, time the execution of these plans, and select the fastest one. Each algorithmic step may break the problem into subproblems, and the fastest plan for each subproblem is constructed in the same way. These timing measurements can either be performed at runtime, or alternatively the plans for a given set of sizes can be precomputed and loaded at a later time.

A direct implementation of this approach, however, faces an exponential explosion of the number of possible plans, and hence of the planning time, as n increases. In order to reduce the planning time to a manageable level, we employ several heuristics to reduce the space of possible plans that must be compared. The most important of these heuristics is **dynamic programming** [\[link\]](#): it optimizes each sub-problem locally, independently of the larger context (so that the “best” plan for a given sub-problem is re-used whenever that sub-problem is encountered). Dynamic programming is not guaranteed to find the fastest plan, because the performance of plans is context-dependent on real machines (e.g., the contents of the cache depend on the preceding computations); however, this approximation works reasonably well in practice and greatly reduces the planning time. Other approximations, such as restrictions on the types of loop-reorderings that are considered "[Plans for higher vector ranks](#)", are described in [\[link\]](#).

Alternatively, there is an **estimate mode** that performs no timing measurements whatsoever, but instead minimizes a heuristic cost function. This can reduce the planner time by several orders of magnitude, but with a significant penalty observed in plan efficiency; e.g., a penalty of 20% is typical for moderate $n \lesssim 2^{13}$, whereas a factor of 2–3 can be suffered for large $n \gtrsim 2^{16}$ [\[link\]](#). Coming up with a better heuristic plan is an interesting open research question; one difficulty is that, because FFT algorithms depend on factorization, knowing a good plan for n does not immediately help one find a good plan for nearby n .

Generating Small FFT Kernels

The base cases of FFTW's recursive plans are its **codelets**, and these form a critical component of FFTW's performance. They consist of long blocks of highly optimized, straight-line code, implementing many special cases of the DFT that give the planner a large space of plans in which to optimize. Not only was it impractical to write numerous codelets by hand, but we also needed to rewrite them many times in order to explore different algorithms and optimizations. Thus, we designed a special-purpose “FFT compiler” called **genfft** that produces the codelets automatically from an abstract description. **genfft** is summarized in this section and described in more detail by [\[link\]](#).

A typical codelet in FFTW computes a DFT of a small, fixed size n (usually, $n \leq 64$), possibly with the input or output multiplied by twiddle factors ["Cooley-Tukey_plans"](#). Several other kinds of codelets can be produced by **genfft**, but we will focus here on this common case.

In principle, all codelets implement some combination of the Cooley-Tukey algorithm from [\[link\]](#) and/or some other DFT algorithm expressed by a similarly compact formula. However, a high-performance implementation of the DFT must address many more concerns than [\[link\]](#) alone suggests. For example, [\[link\]](#) contains multiplications by 1 that are more efficient to omit. [\[link\]](#) entails a run-time factorization of n , which can be precomputed if n is known in advance. [\[link\]](#) operates on complex numbers, but breaking the complex-number abstraction into real and imaginary components turns out to expose certain non-obvious optimizations. Additionally, to exploit the

long pipelines in current processors, the recursion implicit in [\[link\]](#) should be unrolled and re-ordered to a significant degree. Many further optimizations are possible if the complex input is known in advance to be purely real (or imaginary). Our design goal for `genfft` was to keep the expression of the DFT algorithm independent of such concerns. This separation allowed us to experiment with various DFT algorithms and implementation strategies independently and without (much) tedious rewriting.

`genfft` is structured as a compiler whose input consists of the kind and size of the desired codelet, and whose output is C code. `genfft` operates in four phases: creation, simplification, scheduling, and unparsing.

In the **creation** phase, `genfft` produces a representation of the codelet in the form of a directed acyclic graph (**dag**). The dag is produced according to well-known DFT algorithms: Cooley-Tukey [\[link\]](#), prime-factor [\[link\]](#), split-radix [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), and Rader [\[link\]](#). Each algorithm is expressed in a straightforward math-like notation, using complex numbers, with no attempt at optimization. Unlike a normal FFT implementation, however, the algorithms here are evaluated symbolically and the resulting symbolic expression is represented as a dag, and in particular it can be viewed as a **linear network** [\[link\]](#) (in which the edges represent multiplication by constants and the vertices represent additions of the incoming edges).

In the **simplification** phase, `genfft` applies local rewriting rules to each node of the dag in order to simplify it. This phase performs algebraic transformations (such as eliminating multiplications by 1) and common-subexpression elimination. Although such transformations can be performed by a conventional compiler to some degree, they can be carried out here to a greater extent because `genfft` can exploit the specific problem domain. For example, two equivalent subexpressions can always be detected, even if the subexpressions are written in algebraically different forms, because all subexpressions compute linear functions. Also, `genfft` can exploit the property that **network transposition** (reversing the direction of every edge) computes the transposed linear operation [\[link\]](#), in order to transpose the network, simplify, and then transpose back—this turns out to

expose additional common subexpressions [\[link\]](#). In total, these simplifications are sufficiently powerful to derive DFT algorithms specialized for real and/or symmetric data automatically from the complex algorithms. For example, it is known that when the input of a DFT is real (and the output is hence conjugate-symmetric), one can save a little over a factor of two in arithmetic cost by specializing FFT algorithms for this case—with `genfft`, this specialization can be done entirely automatically, pruning the redundant operations from the dag, to match the lowest known operation count for a real-input FFT starting only from the complex-data algorithm [\[link\]](#), [\[link\]](#). We take advantage of this property to help us implement real-data DFTs [\[link\]](#), [\[link\]](#), to exploit machine-specific “SIMD” instructions ["SIMD instructions"](#) [\[link\]](#), and to generate codelets for the discrete cosine (DCT) and sine (DST) transforms [\[link\]](#), [\[link\]](#). Furthermore, by experimentation we have discovered additional simplifications that improve the speed of the generated code. One interesting example is the elimination of negative constants [\[link\]](#): multiplicative constants in FFT algorithms often come in positive/negative pairs, but every C compiler we are aware of will generate separate load instructions for positive and negative versions of the same constants. [\[footnote\]](#) We thus obtained a 10–15% speedup by making all constants positive, which involves propagating minus signs to change additions into subtractions or vice versa elsewhere in the dag (a daunting task if it had to be done manually for tens of thousands of lines of code). Floating-point constants must be stored explicitly in memory; they cannot be embedded directly into the CPU instructions like integer “immediate” constants.

In the **scheduling** phase, `genfft` produces a topological sort of the dag (a **schedule**). The goal of this phase is to find a schedule such that a C compiler can subsequently perform a good register allocation. The scheduling algorithm used by `genfft` offers certain theoretical guarantees because it has its foundations in the theory of cache-oblivious algorithms [\[link\]](#) (here, the registers are viewed as a form of cache), as described in ["Memory strategies in FFTW"](#). As a practical matter, one consequence of this scheduler is that FFTW's machine-independent codelets are no slower than machine-specific codelets generated by SPIRAL [\[link\]](#).

In the stock genfft implementation, the schedule is finally unparsed to C. A variation from [\[link\]](#) implements the rest of a compiler back end and outputs assembly code.

SIMD instructions

Unfortunately, it is impossible to attain nearly peak performance on current popular processors while using only portable C code. Instead, a significant portion of the available computing power can only be accessed by using specialized SIMD (single-instruction multiple data) instructions, which perform the same operation in parallel on a data vector. For example, all modern “x86” processors can execute arithmetic instructions on “vectors” of four single-precision values (SSE instructions) or two double-precision values (SSE2 instructions) at a time, assuming that the operands are arranged consecutively in memory and satisfy a 16-byte alignment constraint. Fortunately, because nearly all of FFTW's low-level code is produced by genfft, machine-specific instructions could be exploited by modifying the generator—the improvements are then automatically propagated to all of FFTW's codelets, and in particular are not limited to a small set of sizes such as powers of two.

SIMD instructions are superficially similar to “vector processors”, which are designed to perform the same operation in parallel on all elements of a data array (a “vector”). The performance of “traditional” vector processors was best for long vectors that are stored in contiguous memory locations, and special algorithms were developed to implement the DFT efficiently on this kind of hardware [\[link\]](#), [\[link\]](#). Unlike in vector processors, however, the SIMD vector length is small and fixed (usually 2 or 4). Because microprocessors depend on caches for performance, one cannot naively use SIMD instructions to simulate a long-vector algorithm: while on vector machines long vectors generally yield better performance, the performance of a microprocessor drops as soon as the data vectors exceed the capacity of the cache. Consequently, SIMD instructions are better seen as a restricted form of instruction-level parallelism than as a degenerate flavor of vector parallelism, and different DFT algorithms are required.

The technique used to exploit SIMD instructions in `genfft` is most easily understood for vectors of length two (e.g., SSE2). In this case, we view a **complex** DFT as a pair of **real** DFTs:

Equation:

$$\text{DFT}(A + i \cdot B) = \text{DFT}(A) + i \cdot \text{DFT}(B) ,$$

where A and B are two real arrays. Our algorithm computes the two real DFTs in parallel using SIMD instructions, and then it combines the two outputs according to [\[link\]](#). This SIMD algorithm has two important properties. First, if the data is stored as an array of complex numbers, as opposed to two separate real and imaginary arrays, the SIMD loads and stores always operate on correctly-aligned contiguous locations, even if the complex numbers themselves have a non-unit stride. Second, because the algorithm finds two-way parallelism in the real and imaginary parts of a single DFT (as opposed to performing two DFTs in parallel), we can completely parallelize DFTs of any size, not just even sizes or powers of 2.

Numerical Accuracy in FFTs

An important consideration in the implementation of any practical numerical algorithm is numerical accuracy: how quickly do floating-point roundoff errors accumulate in the course of the computation? Fortunately, FFT algorithms for the most part have remarkably good accuracy characteristics. In particular, for a DFT of length n computed by a Cooley-Tukey algorithm with finite-precision floating-point arithmetic, the **worst-case** error growth is $O(\log n)$ [\[link\]](#), [\[link\]](#) and the mean error growth for random inputs is only $O(\sqrt{\log n})$ [\[link\]](#), [\[link\]](#). This is so good that, in practical applications, a properly implemented FFT will rarely be a significant contributor to the numerical error.

The amazingly small roundoff errors of FFT algorithms are sometimes explained incorrectly as simply a consequence of the reduced number of operations: since there are fewer operations compared to a naive $O(n^2)$ algorithm, the argument goes, there is less accumulation of roundoff error.

The real reason, however, is more subtle than that, and has to do with the **ordering** of the operations rather than their number. For example, consider the computation of only the output $Y[0]$ in the radix-2 algorithm of [\[link\]](#), ignoring all of the other outputs of the FFT. $Y[0]$ is the sum of all of the inputs, requiring $n - 1$ additions. The FFT does not change this requirement, it merely changes the order of the additions so as to re-use some of them for other outputs. In particular, this radix-2 DIT FFT computes $Y[0]$ as follows: it first sums the even-indexed inputs, then sums the odd-indexed inputs, then adds the two sums; the even- and odd-indexed inputs are summed recursively by the same procedure. This process is sometimes called **cascade summation**, and even though it still requires $n - 1$ total additions to compute $Y[0]$ by itself, its roundoff error grows much more slowly than simply adding $X[0]$, $X[1]$, $X[2]$ and so on in sequence. Specifically, the roundoff error when adding up n floating-point numbers in sequence grows as $O(n)$ in the worst case, or as $O(\sqrt{n})$ on average for random inputs (where the errors grow according to a random walk), but simply reordering these $n-1$ additions into a cascade summation yields $O(\log n)$ worst-case and $O(\sqrt{\log n})$ average-case error growth [\[link\]](#).

However, these encouraging error-growth rates **only** apply if the trigonometric “twiddle” factors in the FFT algorithm are computed very accurately. Many FFT implementations, including FFTW and common manufacturer-optimized libraries, therefore use precomputed tables of twiddle factors calculated by means of standard library functions (which compute trigonometric constants to roughly machine precision). The other common method to compute twiddle factors is to use a trigonometric recurrence formula—this saves memory (and cache), but almost all recurrences have errors that grow as $O(\sqrt{n})$, $O(n)$, or even $O(n^2)$ [\[link\]](#), which lead to corresponding errors in the FFT. For example, one simple recurrence is $e^{i(k+1)\theta} = e^{ik\theta} e^{i\theta}$, multiplying repeatedly by $e^{i\theta}$ to obtain a sequence of equally spaced angles, but the errors when using this process grow as $O(n)$ [\[link\]](#). A common improved recurrence is $e^{i(k+1)\theta} = e^{ik\theta} + e^{ik\theta} (e^{i\theta} - 1)$, where the small quantity [\[footnote\]](#) $e^{i\theta} - 1 = \cos(\theta) - 1 + i \sin(\theta)$ is computed using

$\cos(\theta) - 1 = -2 \sin^2(\theta/2)$ [\[link\]](#); unfortunately, the error using this method still grows as $O(\sqrt{n})$ [\[link\]](#), far worse than logarithmic.

In an FFT, the twiddle factors are powers of ω_n , so θ is a small angle proportional to $1/n$ and $e^{i\theta}$ is close to 1.

There are, in fact, trigonometric recurrences with the same logarithmic error growth as the FFT, but these seem more difficult to implement efficiently; they require that a table of $\Theta(\log n)$ values be stored and updated as the recurrence progresses [\[link\]](#), [\[link\]](#). Instead, in order to gain at least some of the benefits of a trigonometric recurrence (reduced memory pressure at the expense of more arithmetic), FFTW includes several ways to compute a much smaller twiddle table, from which the desired entries can be computed accurately on the fly using a bounded number (usually < 3) of complex multiplications. For example, instead of a twiddle table with n entries ω_n^k , FFTW can use two tables with $\Theta(\sqrt{n})$ entries each, so that ω_n^k is computed by multiplying an entry in one table (indexed with the low-order bits of k) by an entry in the other table (indexed with the high-order bits of k).

There are a few non-Cooley-Tukey algorithms that are known to have worse error characteristics, such as the “real-factor” algorithm [\[link\]](#), [\[link\]](#), but these are rarely used in practice (and are not used at all in FFTW). On the other hand, some commonly used algorithms for type-I and type-IV discrete cosine transforms [\[link\]](#), [\[link\]](#), [\[link\]](#) have errors that we observed to grow as \sqrt{n} even for accurate trigonometric constants (although we are not aware of any theoretical error analysis of these algorithms), and thus we were forced to use alternative algorithms [\[link\]](#).

To measure the accuracy of FFTW, we compare against a slow FFT implemented in arbitrary-precision arithmetic, while to verify the correctness we have found the $O(n \log n)$ self-test algorithm of [\[link\]](#) very useful.

Concluding Remarks

It is unlikely that many readers of this chapter will ever have to implement their own fast Fourier transform software, except as a learning exercise. The computation of the DFT, much like basic linear algebra or integration of ordinary differential equations, is so central to numerical computing and so well-established that robust, flexible, highly optimized libraries are widely available, for the most part as free/open-source software. And yet there are many other problems for which the algorithms are not so finalized, or for which algorithms are published but the implementations are unavailable or of poor quality. Whatever new problems one comes across, there is a good chance that the chasm between theory and efficient implementation will be just as large as it is for FFTs, unless computers become much simpler in the future. For readers who encounter such a problem, we hope that these lessons from FFTW will be useful:

- Generality and portability should almost always come first.
- The number of operations, up to a constant factor, is less important than the order of operations.
- Recursive algorithms with large base cases make optimization easier.
- Optimization, like any tedious task, is best automated.
- Code generation reconciles high-level programming with low-level performance.

We should also mention one final lesson that we haven't discussed in this chapter: you can't optimize in a vacuum, or you end up congratulating yourself for making a slow program slightly faster. We started the FFTW project after downloading a dozen FFT implementations, benchmarking them on a few machines, and noting how the winners varied between machines and between transform sizes. Throughout FFTW's development, we continued to benefit from repeated benchmarks against the dozens of high-quality FFT programs available online, without which we would have thought FFTW was “complete” long ago.

Acknowledgements

SGJ was supported in part by the Materials Research Science and Engineering Center program of the National Science Foundation under award DMR-9400334; MF was supported in part by the Defense Advanced

Research Projects Agency (DARPA) under contract No. NBCH30390004. We are also grateful to Sidney Burrus for the opportunity to contribute this chapter, and for his continual encouragement—dating back to his first kind words in 1997 for the initial FFT efforts of two graduate students venturing outside their fields.

Algorithms for Data with Restrictions

Algorithms for Real Data

Many applications involve processing real data. It is inefficient to simply use a complex FFT on real data because arithmetic would be performed on the zero imaginary parts of the input, and, because of symmetries, output values would be calculated that are redundant. There are several approaches to developing special algorithms or to modifying complex algorithms for real data.

There are two methods which use a complex FFT in a special way to increase efficiency [\[link\]](#), [\[link\]](#). The first method uses a length- N complex FFT to compute two length- N real FFTs by putting the two real data sequences into the real and the imaginary parts of the input to a complex FFT. Because transforms of real data have even real parts and odd imaginary parts, it is possible to separate the transforms of the two inputs with $2N-4$ extra additions. This method requires, however, that two inputs be available at the same time.

The second method [\[link\]](#) uses the fact that the last stage of a decimation-in-time radix-2 FFT combines two independent transforms of length $N/2$ to compute a length- N transform. If the data are real, the two half length transforms are calculated by the method described above and the last stage is carried out to calculate the total length- N FFT of the real data. It should be noted that the half-length FFT does not have to be calculated by a radix-2 FFT. In fact, it should be calculated by the most efficient complex-data algorithm possible, such as the SRFFT or the PFA. The separation of the two half-length transforms and the computation of the last stage requires $N - 6$ real multiplications and $(5/2)N - 6$ real additions [\[link\]](#).

It is possible to derive more efficient real-data algorithms directly rather than using a complex FFT. The basic idea is from Bergland [\[link\]](#), [\[link\]](#) and Sande [\[link\]](#) which, at each stage, uses the symmetries of a constant radix Cooley-Tukey FFT to minimize arithmetic and storage. In the usual derivation [\[link\]](#) of the radix-2 FFT, the length- N transform is written as the combination of the length- $N/2$ DFT of the even indexed data and the

length- $N/2$ DFT of the odd indexed data. If the input to each half-length DFT is real, the output will have Hermitian symmetry. Hence the output of each stage can be arranged so that the results of that stage stores the complex DFT with the real part located where half of the DFT would have gone, and the imaginary part located where the conjugate would have gone. This removes most of the redundant calculations and storage but slightly complicates the addressing. The resulting butterfly structure for this algorithm [\[link\]](#) resembles that for the fast Hartley transform [\[link\]](#). The complete algorithm has one half the number of multiplications and $N-2$ fewer than half the additions of the basic complex FFT. Applying this approach to the split-radix FFT gives a particularly interesting algorithm [\[link\]](#), [\[link\]](#), [\[link\]](#).

Special versions of both the PFA and WFTA can also be developed for real data. Because the operations in the stages of the PFA can be commuted, it is possible to move the combination of the transform of the real part of the input and imaginary part to the last stage. Because the imaginary part of the input is zero, half of the algorithm is simply omitted. This results in the number of multiplications required for the real transform being exactly half of that required for complex data and the number of additions being about N less than half that required for the complex case because adding a pure real number to a pure imaginary number does not require an actual addition. Unfortunately, the indexing and data transfer becomes somewhat more complicated [\[link\]](#), [\[link\]](#). A similar approach can be taken with the WFTA [\[link\]](#), [\[link\]](#), [\[link\]](#).

Special Algorithms for input Data that is mostly Zero, for Calculating only a few Outputs, or where the Sampling is not Uniform

In some cases, most of the data to be transformed are zero. It is clearly wasteful to do arithmetic on that zero data. Another special case is when only a few DFT values are needed. It is likewise wasteful to calculate outputs that are not needed. We use a process called “pruning” to remove the unneeded operations.

In other cases, the data are non-uniform sampling of a continuous time signal [\[link\]](#).

Algorithms for Approximate DFTs

There are applications where approximations to the DFT are all that is needed. [\[link\]](#), [\[link\]](#)

Convolution Algorithms

Fast Convolution by the FFT

One of the main applications of the FFT is to do convolution more efficiently than the direct calculation from the definition which is:

Equation:

$$y(n) = \sum h(m) x(n - m)$$

which, with a change of variables, can also be written as:

Equation:

$$y(n) = \sum x(m) h(n - m)$$

This is often used to filter a signal $x(n)$ with a filter whose impulse response is $h(n)$. Each output value $y(n)$ requires N multiplications and $N - 1$ additions if $y(n)$ and $h(n)$ have N terms. So, for N output values, on the order of N^2 arithmetic operations are required.

Because the DFT converts convolution to multiplication:

Equation:

$$DFT\{y(n)\} = DFT\{h(n)\} DFT\{x(n)\}$$

can be calculated with the FFT and bring the order of arithmetic operations down to $N \log(N)$ which can be significant for large N .

This approach, which is called "fast convolutions", is a form of block processing since a whole block or segment of $x(n)$ must be available to calculate even one output value, $y(n)$. So, a time delay of one block length is always required. Another problem is the filtering use of convolution is usually non-cyclic and the convolution implemented with the DFT is cyclic. This is dealt with by appending zeros to $x(n)$ and $h(n)$ such that the output of the cyclic convolution gives one block of the output of the desired non-cyclic convolution.

For filtering and some other applications, one wants "on going" convolution where the filter response $h(n)$ may be finite in length or duration, but the input $x(n)$ is of arbitrary length. Two methods have traditionally used to break the input into blocks and use the FFT to convolve the block so that the output that would have been calculated by

directly implementing [\[link\]](#) or [\[link\]](#) can be constructed efficiently. These are called “overlap-add” and “over-lap save”.

Fast Convolution by Overlap-Add

In order to use the FFT to convolve (or filter) a long input sequence $x(n)$ with a finite length- M impulse response, $h(n)$, we partition the input sequence in segments or blocks of length L . Because convolution (or filtering) is linear, the output is a linear sum of the result of convolving the first block with $h(n)$ plus the result of convolving the second block with $h(n)$, plus the rest. Each of these block convolutions can be calculated by using the FFT. The output is the inverse FFT of the product of the FFT of $x(n)$ and the FFT of $h(n)$. Since the number of arithmetic operation to calculate the convolution directly is on the order of M^2 and, if done with the FFT, is on the order of $M \log(M)$, there can be a great savings by using the FFT for large M .

The reason this procedure is not totally straightforward, is the length of the output of convolving a length- L block with a length- M filter is of length $L + M - 1$. This means the output blocks cannot simply be concatenated but must be overlapped and added, hence the name for this algorithm is “Overlap-Add”.

The second issue that must be taken into account is the fact that the overlap-add steps need non-cyclic convolution and convolution by the FFT is cyclic. This is easily handled by appending $L - 1$ zeros to the impulse response and $M - 1$ zeros to each input block so that all FFTs are of length $M + L - 1$. This means there is no aliasing and the implemented cyclic convolution gives the same output as the desired non-cyclic convolution.

The savings in arithmetic can be considerable when implementing convolution or performing FIR digital filtering. However, there are two penalties. The use of blocks introduces a delay of one block length. None of the first block of output can be calculated until all of the first block of input is available. This is not a problem for “off line” or “batch” processing but can be serious for real-time processing. The second penalty is the memory required to store and process the blocks. The continuing reduction of memory cost often removes this problem.

The efficiency in terms of number of arithmetic operations per output point increases for large blocks because of the $M \log(M)$ requirements of the FFT. However, the blocks become very large ($L \gg M$), much of the input block will be the appended zeros and efficiency is lost. For any particular application, taking the particular filter and FFT algorithm being used and the particular hardware being used, a plot of efficiency vs. block length, L should be made and L chosen to maximize efficiency given any other constraints that are applicable.

Usually, the block convolutions are done by the FFT, but they could be done by any efficient, finite length method. One could use “rectangular transforms” or “number-theoretic transforms”. A generalization of this method is presented later in the notes.

Fast Convolution by Overlap-Save

An alternative approach to the Overlap-Add can be developed by starting with segmenting the output rather than the input. If one considers the calculation of a block of output, it is seen that not only the corresponding input block is needed, but part of the preceding input block also needed. Indeed, one can show that a length $M + L - 1$ segment of the input is needed for each output block. So, one saves the last part of the preceding block and concatenates it with the current input block, then convolves that with $h(n)$ to calculate the current output

Block Processing, a Generalization of Overlap Methods

Convolution is intimately related to the DFT. It was shown in [The DFT as Convolution or Filtering](#) that a prime length DFT could be converted to cyclic convolution. It has been long known [\[link\]](#) that convolution can be calculated by multiplying the DFTs of signals.

An important question is what is the fastest method for calculating digital convolution. There are several methods that each have some advantage. The earliest method for fast convolution was the use of sectioning with overlap-add or overlap-save and the FFT [\[link\]](#), [\[link\]](#), [\[link\]](#). In most cases the convolution is of real data and, therefore, real-data FFTs should be used. That approach is still probably the fastest method for longer convolution on a general purpose computer or microprocessor. The shorter convolutions should simply be calculated directly.

Introduction

The partitioning of long or infinite strings of data into shorter sections or blocks has been used to allow application of the FFT to realize on-going or continuous convolution [\[link\]](#), [\[link\]](#). This section develops the idea of block processing and shows that it is a generalization of the overlap-add and overlap-save methods [\[link\]](#), [\[link\]](#). They further generalize the idea to a multidimensional formulation of convolution [\[link\]](#), [\[link\]](#). Moving in the opposite direction, it is shown that, rather than partitioning a string of scalars into blocks and then into blocks of blocks, one can partition a scalar number into blocks of bits and then include the operation of multiplication in the signal processing formulation. This is called distributed arithmetic [\[link\]](#) and, since it describes operations

at the bit level, is completely general. These notes try to present a coherent development of these ideas.

Block Signal Processing

In this section the usual convolution and recursion that implements FIR and IIR discrete-time filters are reformulated in terms of vectors and matrices. Because the same data is partitioned and grouped in a variety of ways, it is important to have a consistent notation in order to be clear. The n^{th} element of a data sequence is expressed $h(n)$ or, in some cases to simplify, h_n . A block or finite length column vector is denoted h_n with n indicating the n^{th} block or section of a longer vector. A matrix, square or rectangular, is indicated by an upper case letter such as H with a subscript if appropriate.

Block Convolution

The operation of a finite impulse response (FIR) filter is described by a finite convolution as

Equation:

$$y(n) = \sum_{k=0}^{L-1} h(k) x(n-k)$$

where $x(n)$ is causal, $h(n)$ is causal and of length L , and the time index n goes from zero to infinity or some large value. With a change of index variables this becomes

Equation:

$$y(n) = \sum h(n-k) x(k)$$

which can be expressed as a matrix operation by

Equation:

$$\begin{array}{ccccccc} y_0 & & h_0 & 0 & 0 & \cdots & 0 & x_0 \\ y_1 & & h_1 & h_0 & 0 & & & x_1 \\ y_2 & = & h_2 & h_1 & h_0 & & & x_2 \\ \vdots & & \vdots & & & & \vdots & \vdots \end{array} \cdot$$

The H matrix of impulse response values is partitioned into N by N square sub matrices and the X and Y vectors are partitioned into length- N blocks or sections. This is illustrated for $N = 3$ by

Equation:

$$H_0 = \begin{bmatrix} h_0 & 0 & 0 \\ h_1 & h_0 & 0 \\ h_2 & h_1 & h_0 \end{bmatrix} \quad H_1 = \begin{bmatrix} h_3 & h_2 & h_1 \\ h_4 & h_3 & h_2 \\ h_5 & h_4 & h_3 \end{bmatrix} \quad \text{etc.}$$

Equation:

$$\begin{array}{ccc} x_0 & x_3 & y_0 \\ x_0 = x_1 & x_1 = x_4 & \underline{y}_0 = y_1 \quad \text{etc.} \\ x_2 & x_5 & y_2 \end{array}$$

Substituting these definitions into [\[link\]](#) gives

Equation:

$$\begin{array}{ccccccc} \underline{y}_0 & H_0 & 0 & 0 & \cdots & 0 & x_0 \\ \underline{y}_1 & H_1 & H_0 & 0 & & & x_1 \\ \underline{y}_2 & H_2 & H_1 & H_0 & & & x_2 \\ \vdots & \vdots & & & & \vdots & \vdots \end{array} =$$

The general expression for the n^{th} output block is

Equation:

$$\underline{y}_n = \sum_{k=0}^n H_{n-k} x_k$$

which is a vector or block convolution. Since the matrix-vector multiplication within the block convolution is itself a convolution, [\[link\]](#) is a sort of convolution of convolutions and the finite length matrix-vector multiplication can be carried out using the FFT or other fast convolution methods.

The equation for one output block can be written as the product

Equation:

$$\underline{y}_2 = \begin{bmatrix} H_2 H_1 H_0 \end{bmatrix} \begin{matrix} x_0 \\ x_1 \\ x_2 \end{matrix}$$

and the effects of one input block can be written

Equation:

$$\begin{matrix} H_0 \\ H_1 \\ H_2 \end{matrix} \begin{matrix} x_0 \\ x_1 \\ x_2 \end{matrix} = \begin{matrix} y_0 \\ y_1 \\ y_2 \end{matrix} .$$

These are generalize statements of overlap save and overlap add [\[link\]](#), [\[link\]](#). The block length can be longer, shorter, or equal to the filter length.

Block Recursion

Although less well-known, IIR filters can also be implemented with block processing [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#). The block form of an IIR filter is developed in much the same way as for the block convolution implementation of the FIR filter. The general constant coefficient difference equation which describes an IIR filter with recursive coefficients a_l , convolution coefficients b_k , input signal $x(n)$, and output signal $y(n)$ is given by

Equation:

$$y(n) = \sum_{l=1}^{N-1} a_l y_{n-l} + \sum_{k=0}^{M-1} b_k x_{n-k}$$

using both functional notation and subscripts, depending on which is easier and clearer. The impulse response $h(n)$ is

Equation:

$$h(n) = \sum_{l=1}^{N-1} a_l h(n-l) + \sum_{k=0}^{M-1} b_k \delta(n-k)$$

which can be written in matrix operator form

Equation:

$$\begin{array}{cccccc}
 1 & 0 & 0 & \cdots & 0 & h_0 & b_0 \\
 a_1 & 1 & 0 & & & h_1 & b_1 \\
 a_2 & a_1 & 1 & & & h_2 & b_2 \\
 a_3 & a_2 & a_1 & & & h_3 & = & b_3 \\
 0 & a_3 & a_2 & & & h_4 & 0 \\
 \vdots & & & & \vdots & \vdots & \vdots
 \end{array}$$

In terms of N by N submatrices and length- N blocks, this becomes

Equation:

$$\begin{array}{cccccc}
 A_0 & 0 & 0 & \cdots & 0 & h_0 & \underline{b}_0 \\
 A_1 & A_0 & 0 & & & h_1 & \underline{b}_1 \\
 0 & A_1 & A_0 & & & h_2 & = & 0 \\
 \vdots & & & & \vdots & \vdots & \vdots
 \end{array}$$

From this formulation, a block recursive equation can be written that will generate the impulse response block by block.

Equation:

$$A_0 h_n + A_1 h_{n-1} = 0 \text{ for } n \geq 2$$

Equation:

$$h_n = -A_0^{-1} A_1 h_{n-1} = K h_{n-1} \text{ for } n \geq 2$$

with initial conditions given by

Equation:

$$h_1 = -A_0^{-1} A_1 A_0^{-1} \underline{b}_0 + A_0^{-1} \underline{b}_1$$

This can also be written to generate the square partitions of the impulse response matrix by

Equation:

$$H_n = K H_{n-1} \text{ for } n \geq 2$$

with initial conditions given by

Equation:

$$H_1 = K A_0^{-1} B_0 + A_0^{-1} B_1$$

and $K = -A_0^{-1} A_1$. This recursively generates square submatrices of H similar to those defined in [\[link\]](#) and [\[link\]](#) and shows the basic structure of the dynamic system.

Next, we develop the recursive formulation for a general input as described by the scalar difference equation [\[link\]](#) and in matrix operator form by

Equation:

$$\begin{array}{cccccc} 1 & 0 & 0 & \cdots & 0 & y_0 & b_0 & 0 & 0 & \cdots & 0 & x_0 \\ a_1 & 1 & 0 & & & y_1 & b_1 & b_0 & 0 & & & x_1 \\ a_2 & a_1 & 1 & & & y_2 & b_2 & b_1 & b_0 & & & x_2 \\ a_3 & a_2 & a_1 & & & y_3 & 0 & b_2 & b_1 & & & x_3 \\ 0 & a_3 & a_2 & & & y_4 & 0 & 0 & b_2 & & & x_4 \\ \vdots & & & & \vdots & \vdots & \vdots & & & \vdots & \vdots & \end{array} =$$

which, after substituting the definitions of the sub matrices and assuming the block length is larger than the order of the numerator or denominator, becomes

Equation:

$$\begin{array}{cccccc} A_0 & 0 & 0 & \cdots & 0 & \underline{y}_0 & B_0 & 0 & 0 & \cdots & 0 & x_0 \\ A_1 & A_0 & 0 & & & \underline{y}_1 & B_1 & B_0 & 0 & & & x_1 \\ 0 & A_1 & A_0 & & & \underline{y}_2 & 0 & B_1 & B_0 & & & x_2 \\ \vdots & & & & \vdots & \vdots & \vdots & & & \vdots & \vdots & \end{array} =$$

From the partitioned rows of [\[link\]](#), one can write the block recursive relation

Equation:

$$A_0 \underline{y}_{n+1} + A_1 \underline{y}_n = B_0 x_{n+1} + B_1 x_n$$

Solving for \underline{y}_{n+1} gives

Equation:

$$\underline{y}_{n+1} = -A_0^{-1} A_1 \underline{y}_n + A_0^{-1} B_0 x_{n+1} + A_0^{-1} B_1 x_n$$

Equation:

$$\underline{y}_{n+1} = K \underline{y}_n + H_0 x_{n+1} + \tilde{H}_1 x_n$$

which is a first order vector difference equation [\[link\]](#), [\[link\]](#). This is the fundamental block recursive algorithm that implements the original scalar difference equation in [\[link\]](#). It has several important characteristics.

- The block recursive formulation is similar to a state variable equation but the states are blocks or sections of the output [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#).
- The eigenvalues of K are the poles of the original scalar problem raised to the N power plus others that are zero. The longer the block length, the “more stable” the filter is, i.e. the further the poles are from the unit circle [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#).
- If the block length were shorter than the denominator, the vector difference equation would be higher than first order. There would be a non zero A_2 . If the block length were shorter than the numerator, there would be a non zero B_2 and a higher order block convolution operation. If the block length were one, the order of the vector equation would be the same as the scalar equation. They would be the same equation.
- The actual arithmetic that goes into the calculation of the output is partly recursive and partly convolution. The longer the block, the more the output is calculated by convolution and, the more arithmetic is required.
- It is possible to remove the zero eigenvalues in K by making K rectangular or square and N by N . This results in a form even more similar to a state variable formulation [\[link\]](#), [\[link\]](#). This is briefly discussed below in section 2.3.
- There are several ways of using the FFT in the calculation of the various matrix products in [\[link\]](#) and in [\[link\]](#) and [\[link\]](#). Each has some arithmetic advantage for various forms and orders of the original equation. It is also possible to implement some of the operations using rectangular transforms, number theoretic transforms, distributed arithmetic, or other efficient convolution algorithms [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#).
- By choosing the block length equal to the period, a periodically time varying filter can be made block time invariant. In other words, all the time varying characteristics are moved to the finite matrix multiplies which leave the time invariant properties at the block level. This allows use of z-transform and other time-invariant methods to be used for stability analysis and frequency response analysis [\[link\]](#), [\[link\]](#). It also turns out to be related to filter banks and multi-rate filters [\[link\]](#), [\[link\]](#), [\[link\]](#).

Block State Formulation

It is possible to reduce the size of the matrix operators in the block recursive description [\[link\]](#) to give a form even more like a state variable equation [\[link\]](#), [\[link\]](#), [\[link\]](#). If K in [\[link\]](#) has several zero eigenvalues, it should be possible to reduce the size of K until it has full rank. That was done in [\[link\]](#) and the result is

Equation:

$$\underline{z}_n = K_1 \underline{z}_{n-1} + K_2 x_n$$

Equation:

$$\underline{y}_n = H_1 \underline{z}_{n-1} + H_0 x_n$$

where H_0 is the same N by N convolution matrix, N_1 is a rectangular L by N partition of the convolution matrix H , K_1 is a square N by N matrix of full rank, and K_2 is a rectangular N by L matrix.

This is now a minimal state equation whose input and output are blocks of the original input and output. Some of the matrix multiplications can be carried out using the FFT or other techniques.

Block Implementations of Digital Filters

The advantage of the block convolution and recursion implementations is a possible improvement in arithmetic efficiency by using the FFT or other fast convolution methods for some of the multiplications in [\[link\]](#) or [\[link\]](#) [\[link\]](#), [\[link\]](#). There is the reduction of quantization effects due to an effective decrease in the magnitude of the eigenvalues and the possibility of easier parallel implementation for IIR filters. The disadvantages are a delay of at least one block length and an increased memory requirement.

These methods could also be used in the various filtering methods for evaluating the DFT. This the chirp z-transform, Rader's method, and Goertzel's algorithm.

Multidimensional Formulation

This process of partitioning the data vectors and the operator matrices can be continued by partitioning [\[link\]](#) and [\[link\]](#) and creating blocks of blocks to give a higher

dimensional structure. One should use index mapping ideas rather than partitioned matrices for this approach [\[link\]](#), [\[link\]](#).

Periodically Time-Varying Discrete-Time Systems

Most time-varying systems are periodically time-varying and this allows special results to be obtained. If the block length is set equal to the period of the time variations, the resulting block equations are time invariant and all to the time varying characteristics are contained in the matrix multiplications. This allows some of the tools of time invariant systems to be used on periodically time-varying systems.

The PTV system is analyzed in [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), the filter analysis and design problem, which includes the decimation–interpolation structure, is addressed in [\[link\]](#), [\[link\]](#), [\[link\]](#), and the bandwidth compression problem in [\[link\]](#). These structures can take the form of filter banks [\[link\]](#).

Multirate Filters, Filter Banks, and Wavelets

Another area that is related to periodically time varying systems and to block processing is filter banks [\[link\]](#), [\[link\]](#). Recently the area of perfect reconstruction filter banks has been further developed and shown to be closely related to wavelet based signal analysis [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#). The filter bank structure has several forms with the polyphase and lattice being particularly interesting.

An idea that has some elements of multirate filters, perfect reconstruction, and distributed arithmetic is given in [\[link\]](#), [\[link\]](#), [\[link\]](#). Parks has noted that design of multirate filters has some elements in common with complex approximation and of 2-D filter design [\[link\]](#), [\[link\]](#) and is looking at using Tang's method for these designs.

Distributed Arithmetic

Rather than grouping the individual scalar data values in a discrete-time signal into blocks, the scalar values can be partitioned into groups of bits. Because multiplication of integers, multiplication of polynomials, and discrete-time convolution are the same operations, the bit-level description of multiplication can be mixed with the convolution of the signal processing. The resulting structure is called distributed arithmetic [\[link\]](#), [\[link\]](#). It can be used to create an efficient table look-up scheme to implement an FIR or IIR filter using no multiplications by fetching previously calculated partial products which are stored in a table. Distributed arithmetic, block processing, and multi-

dimensional formulations can be combined into an integrated powerful description to implement digital filters and processors. There may be a new form of distributed arithmetic using the ideas in [\[link\]](#), [\[link\]](#).

Direct Fast Convolution and Rectangular Transforms

A relatively new approach uses index mapping directly to convert a one dimensional convolution into a multidimensional convolution [\[link\]](#), [\[link\]](#). This can be done by either a type-1 or type-2 map. The short convolutions along each dimension are then done by Winograd's optimal algorithms. Unlike for the case of the DFT, there is no savings of arithmetic from the index mapping alone. All the savings comes from efficient short algorithms. In the case of index mapping with convolution, the multiplications must be nested together in the center of the algorithm in the same way as for the WFTA. There is no equivalent to the PFA structure for convolution. The multidimensional convolution can not be calculated by row and column convolutions as the DFT was by row and column DFTs.

It would first seem that applying the index mapping and optimal short algorithms directly to convolution would be more efficient than using DFTs and converting them to convolution to be calculated by the same optimal algorithms. In practical algorithms, however, the DFT method seems to be more efficient [\[link\]](#).

A method that is attractive for special purpose hardware uses distributed arithmetic [\[link\]](#). This approach uses a table look up of precomputed partial products to produce a system that does convolution without requiring multiplications [\[link\]](#).

Another method that requires special hardware uses number theoretic transforms [\[link\]](#), [\[link\]](#), [\[link\]](#) to calculate convolution. These transforms are defined over finite fields or rings with arithmetic performed modulo special numbers. These transforms have rather limited flexibility, but when they can be used, they are very efficient.

Number Theoretic Transforms for Convolution

Results from Number Theory

A basic review of the number theory useful for signal processing algorithms will be given here with specific emphasis on the congruence theory for number theoretic transforms [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#).

Number Theoretic Transforms

Here we look at the conditions placed on a general linear transform in order for it to support cyclic convolution. The form of a linear transformation of a length-N sequence of number is given by

Equation:

$$X(k) = \sum_{n=0}^{N-1} t(n, k) x(n)$$

for $k = 0, 1, \dots, (N - 1)$. The definition of cyclic convolution of two sequences is given by

Equation:

$$y(n) = \sum_{m=0}^{N-1} x(m) h(n - m)$$

for $n = 0, 1, \dots, (N - 1)$ and all indices evaluated modulo N. We would like to find the properties of the transformation such that it will support the cyclic convolution. This means that if $X(k)$, $H(k)$, and $Y(k)$ are the transforms of $x(n)$, $h(n)$, and $y(n)$ respectively,

Equation:

$$Y(k) = X(k) H(k).$$

The conditions are derived by taking the transform defined in [\[link\]](#) of both sides of equation [\[link\]](#) which gives

Equation:

$$Y(k) = \sum_{n=0}^{N-1} t(n, k) \sum_{m=0}^{N-1} x(m) h(n - m)$$

Equation:

$$= \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x(m) h(n - m) t(n, k).$$

Making the change of index variables, $l = n - m$, gives

Equation:

$$= \sum_{m=0}^{N-1} \sum_{l=0}^{N-1} x(m) h(l) t(l+m, k).$$

But from [\[link\]](#), this must be

Equation:

$$Y(k) = \sum_{n=0}^{N-1} x(n) t(n, k) \sum_{m=0}^{N-1} x(m) t(m, k)$$

Equation:

$$= \sum_{m=0}^{N-1} \sum_{l=0}^{N-1} x(m) h(l) t(n, k) t(l, k).$$

This must be true for all $x(n)$, $h(n)$, and k , therefore from [\[link\]](#) and [\[link\]](#) we have

Equation:

$$t(m+l, k) = t(m, k) t(l, k)$$

For $l = 0$ we have

Equation:

$$t(m, k) = t(m, k) t(0, k)$$

and, therefore, $t(0, k) = 1$. For $l = m$ we have

Equation:

$$t(2m, k) = t(m, k) t(m, k) = t^2(m, k)$$

For $l = pm$ we likewise have

Equation:

$$t(pm, k) = t^p(m, k)$$

and, therefore,

Equation:

$$t^N(m, k) = t(Nm, k) = t(0, k) = 1.$$

But

Equation:

$$t(m, k) = t^m(1, k) = t^k(m, 1),$$

therefore,

Equation:

$$t(m, k) = t^{mk}(1, 1).$$

Defining $t(1, 1) = \alpha$ gives the form for our general linear transform [\[link\]](#) as

Equation:

$$X(k) = \sum_{n=0}^{N-1} \alpha^{nk} x(n)$$

where α is a root of order N , which means that N is the smallest integer such that $\alpha^N = 1$.

Theorem 1 The transform [\[link\]](#) supports cyclic convolution if and only if α is a root of order N and N^{-1} is defined.

This is discussed in [\[link\]](#), [\[link\]](#).

Theorem 2 The transform [\[link\]](#) supports cyclic convolution if and only if

Equation:

$$N | O(M)$$

where

Equation:

$$O(M) = \gcd\{p_1 - 1, p_2 - 1, \dots, p_l - 1\}$$

and

Equation:

t	b	$M = F_t$	N_2	$N_{\sqrt{2}}$	N_{max}	α for N_{max}
3	8	$2^8 + 1$	16	32	256	3
4	16	$2^{16} + 1$	32	64	65536	3
5	32	$2^{32} + 1$	64	128	128	$\sqrt{2}$
6	64	$2^{64} + 1$	128	256	256	$\sqrt{2}$

This table gives values of N for the two most important values of α which are 2 and $\sqrt{2}$. The second column give the approximate number of bits in the number representation. The third column gives the Fermat number modulus, the fourth is the maximum convolution length for $\alpha = 2$, the fifth is the maximum length for $\alpha = \sqrt{2}$, the sixth is the maximum length for any α , and the seventh is the α for that maximum length. Remember that the first two rows have a Fermat number modulus which is prime and second two rows have a composite Fermat number as modulus. Note the differences.

The books, articles, and presentations that discuss NTT and related topics are [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#). A recent book discusses NT in a signal processing context [\[link\]](#).

Other work and Results

This section comes from a note describing results on efficient algorithms to calculate the discrete Fourier transform (DFT) that were collected over years. Perhaps the most interesting is the discovery that the Cooley-Tukey FFT was described by Gauss in 1805 [\[link\]](#). That gives some indication of the age of research on the topic, and the fact that a 1995 compiled bibliography [\[link\]](#) on efficient algorithms contains over 3400 entries indicates its volume. Three IEEE Press reprint books contain papers on the FFT [\[link\]](#), [\[link\]](#), [\[link\]](#). An excellent general purpose FFT program has been described in [\[link\]](#), [\[link\]](#) and is used in Matlab and available over the internet.

In addition to this book there are several others [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#) that give a good modern theoretical background for the FFT, one book [\[link\]](#) that gives the basic theory plus both FORTRAN and TMS 320 assembly language programs, and other books [\[link\]](#), [\[link\]](#), [\[link\]](#) that contain chapters on advanced FFT topics. A good up-to-date, on-line reference with both theory and programming techniques is in [\[link\]](#). The history of the FFT is outlined in [\[link\]](#), [\[link\]](#) and excellent survey articles can be found in [\[link\]](#), [\[link\]](#). The foundation of much of the modern work on efficient algorithms was done by S. Winograd. These results can be found in [\[link\]](#), [\[link\]](#), [\[link\]](#). An outline and discussion of his theorems can be found in [\[link\]](#) as well as [\[link\]](#), [\[link\]](#), [\[link\]](#).

Efficient FFT algorithms for length- 2^M were described by Gauss and discovered in modern times by Cooley and Tukey [\[link\]](#). These have been highly developed and good examples of FORTRAN programs can be found in [\[link\]](#). Several new algorithms have been published that require the least known amount of total arithmetic [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#). Of these, the split-radix FFT [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#) seems to have the best structure for programming, and an efficient program has been written [\[link\]](#) to implement it. A mixture of decimation-in-time and decimation-in-frequency with very good efficiency is given in [\[link\]](#), [\[link\]](#) and one called

the Sine-Cosine FT [\[link\]](#). Recently a modification to the split-radix algorithm has been described [\[link\]](#) that has a slightly better total arithmetic count. Theoretical bounds on the number of multiplications required for the FFT based on Winograd's theories are given in [\[link\]](#), [\[link\]](#). Schemes for calculating an in-place, in-order radix-2 FFT are given in [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#). Discussion of various forms of unscramblers is given in [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#). A discussion of the relation of the computer architecture, algorithm and compiler can be found in [\[link\]](#), [\[link\]](#). A modification to allow lengths of $N = q 2^m$ for q odd is given in [\[link\]](#).

The “other” FFT is the prime factor algorithm (PFA) which uses an index map originally developed by Thomas and by Good. The theory of the PFA was derived in [\[link\]](#) and further developed and an efficient in-order and in-place program given in [\[link\]](#), [\[link\]](#). More results on the PFA are given in [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#), [\[link\]](#). A method has been developed to use dynamic programming to design optimal FFT programs that minimize the number of additions and data transfers as well as multiplications [\[link\]](#). This new approach designs custom algorithms for a particular computer architecture. An efficient and practical development of Winograd's ideas has given a design method that does not require the rather difficult Chinese remainder theorem [\[link\]](#), [\[link\]](#) for short prime length FFT's. These ideas have been used to design modules of length 11, 13, 17, 19, and 25 [\[link\]](#). Other methods for designing short DFT's can be found in [\[link\]](#), [\[link\]](#). A use of these ideas with distributed arithmetic and table look-up rather than multiplication is given in [\[link\]](#). A program that implements the nested Winograd Fourier transform algorithm (WFTA) is given in [\[link\]](#) but it has not proven as fast or as versatile as the PFA [\[link\]](#). An interesting use of the PFA was announced [\[link\]](#) in searching for large prime numbers.

These efficient algorithms can not only be used on DFT's but on other transforms with a similar structure. They have been applied to the discrete Hartley transform [\[link\]](#), [\[link\]](#) and the discrete cosine transform [\[link\]](#), [\[link\]](#), [\[link\]](#).

The fast Hartley transform has been proposed as a superior method for real data analysis but that has been shown not to be the case. A well-designed

Various approaches to calculating approximate DFTs have been based on cordic methods, short word lengths, or some form of pruning. A new method that uses the characteristics of the signals being transformed has combined the discrete wavelet transform (DWT) combined with the DFT to give an approximate FFT with $O(N)$ multiplications [\[link\]](#), [\[link\]](#), [\[link\]](#) for certain signal classes. A similar approach has been developed using filter banks [\[link\]](#), [\[link\]](#).

The study of efficient algorithms not only has a long history and large bibliography, it is still an exciting research field where new results are used in practical applications.

More information can be found on the [Rice DSP Group's web page](#)

Conclusions: Fast Fourier Transforms

This book has developed a class of efficient algorithms based on index mapping and polynomial algebra. This provides a framework from which the Cooley-Tukey FFT, the split-radix FFT, the PFA, and WFTA can be derived. Even the programs implementing these algorithms can have a similar structure. Winograd's theorems were presented and shown to be very powerful in both deriving algorithms and in evaluating them. The simple radix-2 FFT provides a compact, elegant means for efficiently calculating the DFT. If some elaboration is allowed, significant improvement can be had from the split-radix FFT, the radix-4 FFT or the PFA. If multiplications are expensive, the WFTA requires the least of all.

Several methods for transforming real data were described that are more efficient than directly using a complex FFT. A complex FFT can be used for real data by artificially creating a complex input from two sections of real input. An alternative and slightly more efficient method is to construct a special FFT that utilizes the symmetries at each stage.

As computers move to multiprocessors and multicore, writing and maintaining efficient programs becomes more and more difficult. The highly structured form of FFTs allows automatic generation of very efficient programs that are tailored specifically to a particular DSP or computer architecture.

For high-speed convolution, the traditional use of the FFT or PFA with blocking is probably the fastest method although rectangular transforms, distributed arithmetic, or number theoretic transforms may have a future with special VLSI hardware.

The ideas presented in these notes can also be applied to the calculation of the discrete Hartley transform [\[link\]](#), [\[link\]](#), the discrete cosine transform [\[link\]](#), [\[link\]](#), and to number theoretic transforms [\[link\]](#), [\[link\]](#), [\[link\]](#).

There are many areas for future research. The relationship of hardware to algorithms, the proper use of multiple processors, the proper design and use of array processors and vector processors are all open. There are still many

unanswered questions in multi-dimensional algorithms where a simple extension of one-dimensional methods will not suffice.

Appendix 1: FFT Flowgraphs

Flowgraphs of various radix-2 and 4 Cooley Tukey FFTs and Split Radix FFTs.

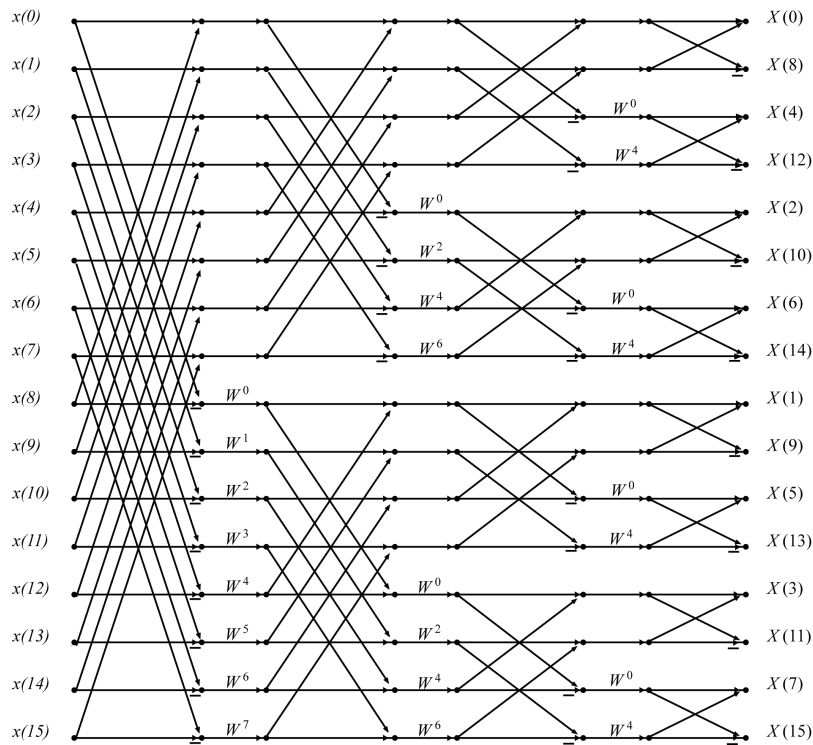
Signal Flow Graphs of Cooley-Tukey FFTs

The following four figures are flow graphs for Radix-2 Cooley-Tukey FFTs. The first is a length-16, decimation-in-frequency Radix-2 FFT with the input data in order and output data scrambled. The first stage has 8 length-2 "butterflies" (which overlap in the figure) followed by 8 multiplications by powers of W which are called "twiddle factors". The second stage has 2 length-8 FFTs which are each calculated by 4 butterflies followed by 4 multiplies. The third stage has 4 length-4 FFTs, each calculated by 2 butterflies followed by 2 multiplies and the last stage is simply 8 butterflies followed by trivial multiplies by one. This flow graph should be compared with the index map in [Polynomial Description of Signals](#), the polynomial decomposition in [The DFT as Convolution or Filtering](#), and the program in [Appendix 3](#). In the program, the butterflies and twiddle factor multiplications are done together in the inner most loop. The outer most loop indexes through the stages. If the length of the FFT is a power of two, the number of stages is that power ($\log N$).

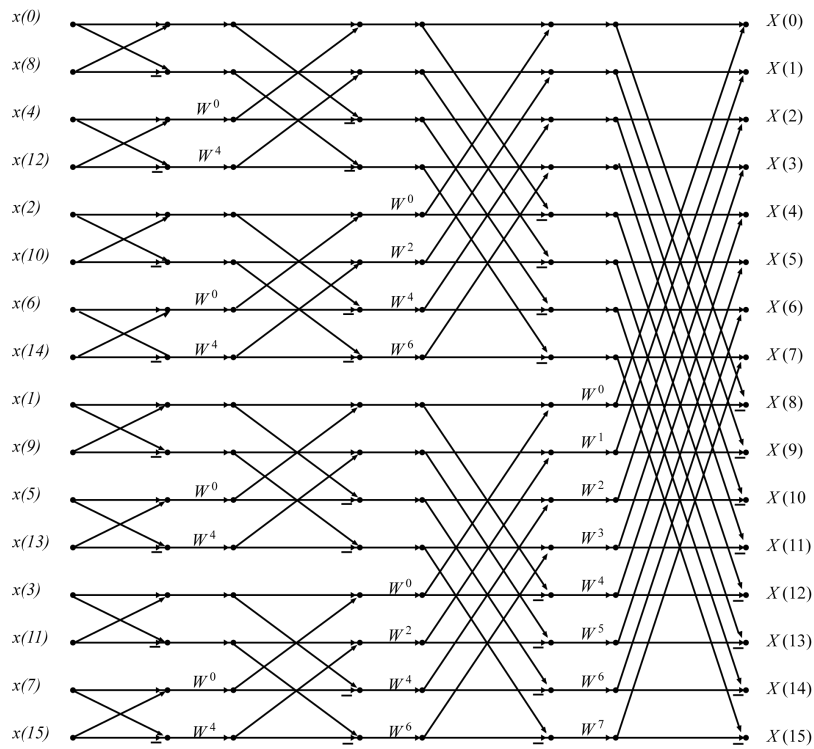
The second figure below is a length-16, decimation-in-time FFT with the input data scrambled and output data in order. The first stage has 8 length-2 "butterflies" followed by 8 twiddle factors multiplications. The second stage has 4 length-4 FFTs which are each calculated by 2 butterflies followed by 2 multiplies. The third stage has 2 length-8 FFTs, each calculated by 4 butterflies followed by 8 multiplies and the last stage is simply 8 length-2 butterflies. This flow graph should be compared with the index map in [Polynomial Description of Signals](#), the polynomial decomposition in [The DFT as Convolution or Filtering](#), and the program in [Appendix 3](#). Here, the FFT must be preceded by a scrambler.

The third and fourth figures below are a length-16 decimation-in-frequency and a decimation-in-time but, in contrast to the figures above, the DIF has the output in order which requires a scrambled input and the DIT has the input in order which requires the output be unscrambled. Compare with the

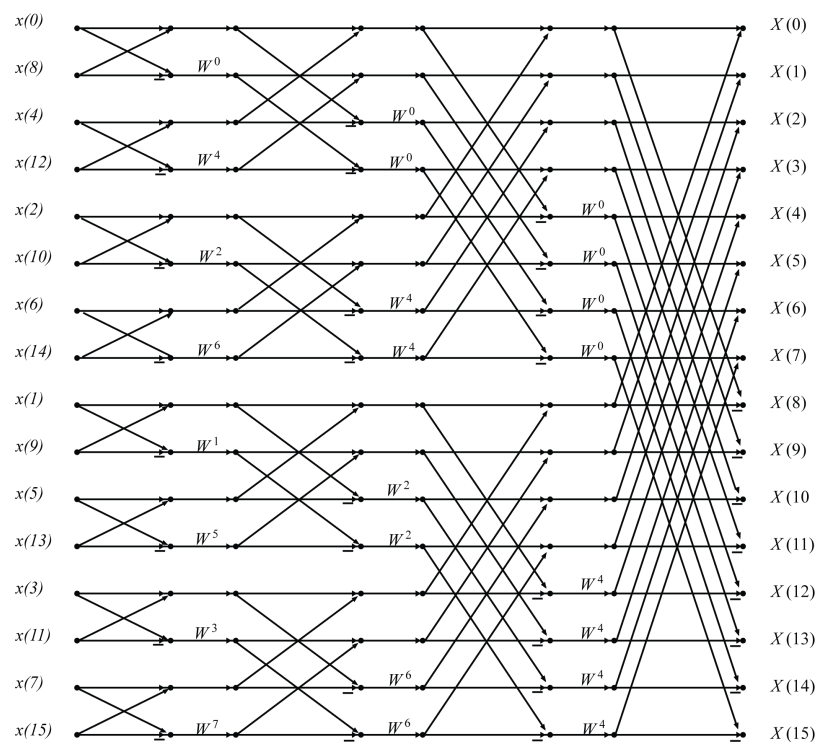
first two figures. Note the order of the twiddle factors. The number of additions and multiplications in all four flow graphs is the same and the structure of the three-loop program which executes the flow graph is the same.



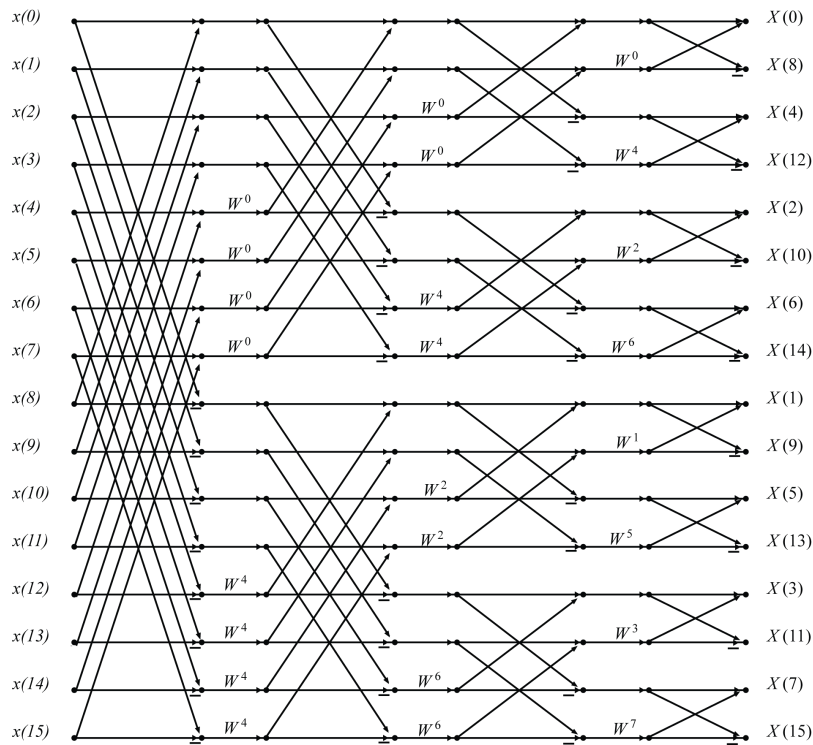
Length-16, Decimation-in-Frequency, In-order input, Radix-2 FFT



Length-16, Decimation-in-Time, In-order
output, Radix-2 FFT

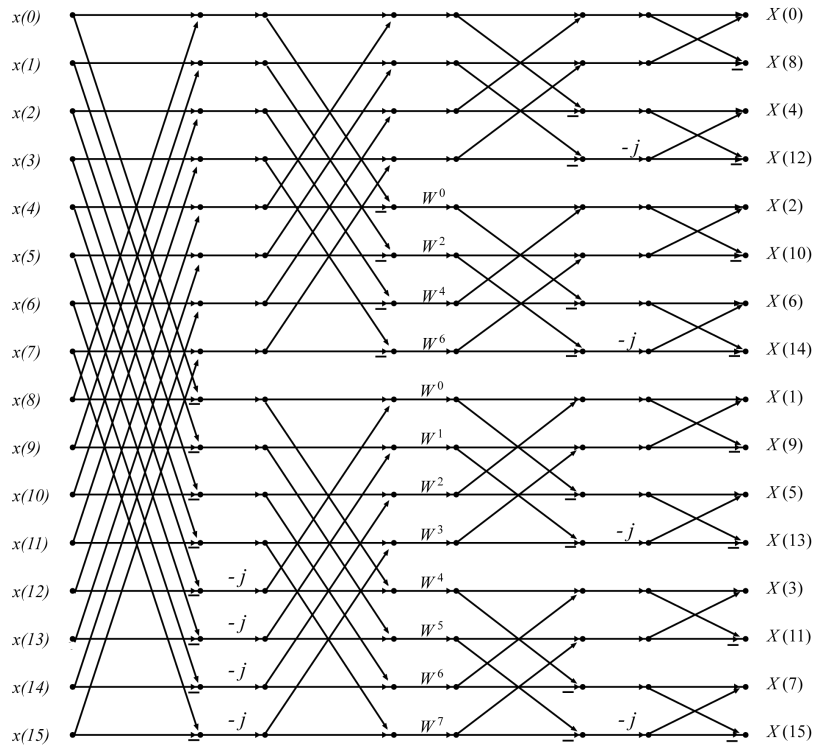


Length-16, alternate Decimation-in-Frequency, In-order output, Radix-2 FFT



Length-16, alternate Decimation-in-Time,
In-order input, Radix-2 FFT

The following is a length-16, decimation-in-frequency Radix-4 FFT with the input data in order and output data scrambled. There are two stages with the first stage having 4 length-4 "butterflies" followed by 12 multiplications by powers of W which are called "twiddle factors". The second stage has 4 length-4 FFTs which are each calculated by 4 butterflies followed by 4 multiplies. Note, each stage here looks like two stages but it is one and there is only one place where twiddle factor multiplications appear. This flow graph should be compared with the index map in [Polynomial Description of Signals](#), the polynomial decomposition in [The DFT as Convolution or Filtering](#), and the program in [Appendix 3](#). Log to the base 4 of 16 is 2. The total number of twiddle factor multiplication here is 12 compared to 24 for the radix-2. The unscrambler is a base-four reverse order counter rather than a bit reverse counter, however, a modification of the radix four butterflies will allow a bit reverse counter to be used with the radix-4 FFT as with the radix-2.

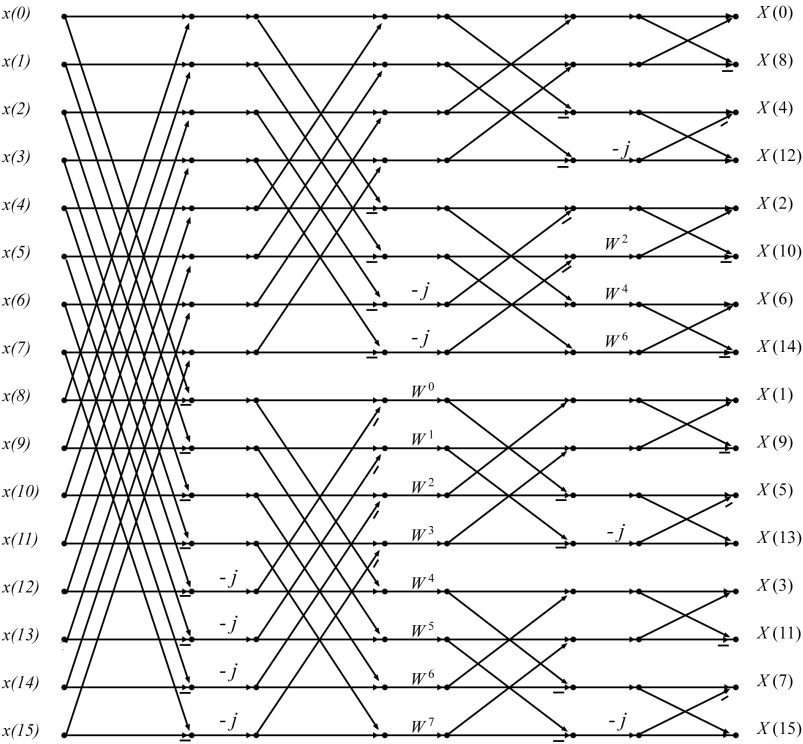


Length-16, Decimation-in-Frequency, In-order input, Radix-4 FFT

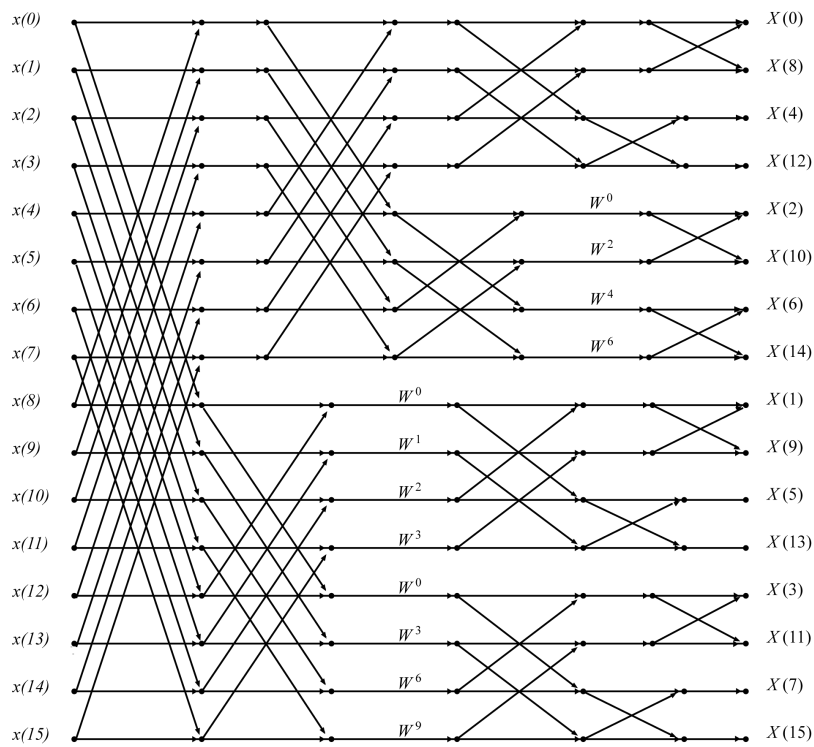
The following two flowgraphs are length-16, decimation-in-frequency Split Radix FFTs with the input data in order and output data scrambled. Because the "butterflies" are L shaped, the stages do not progress uniformly like the Radix-2 or 4. These two figures are the same with the first drawn in a way to compare with the Radix-2 and 4, and the second to illustrate the L shaped butterflies. These flow graphs should be compared with the index map in [Polynomial Description of Signals](#) and the program in [Appendix 3](#). Because of the non-uniform stages, the program indexing is more complicated. Although the number of twiddle factor multiplications is 12 as was the radix-4 case, for longer lengths, the split-radix has slightly fewer multiplications than the radix-4.

Because the structures of the radix-2, radix-4, and split-radix FFTs are the same, the number of data additions is same for all of them. However, each complex twiddle factor multiplication requires two real additions (and four

real multiplications) the number of additions will be fewer for the structures with fewer multiplications.



Length-16, Decimation-in-Frequency, In-order input, Split-Radix FFT



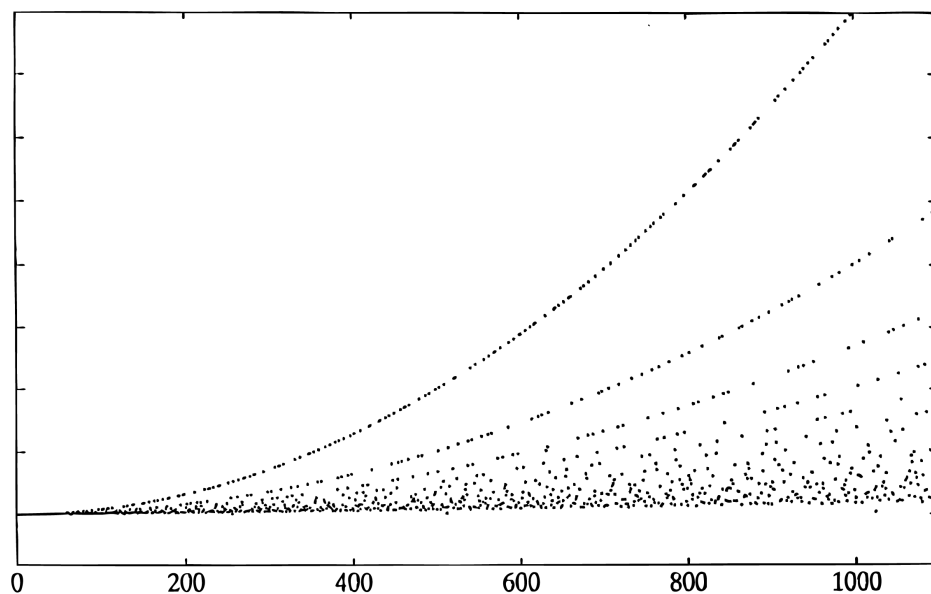
Length-16, Decimation-in-Frequency, Split-Radix with special BFs FFT

Appendix 2: Operation Counts for General Length FFT

Operation Counts for General Length FFT

Figures

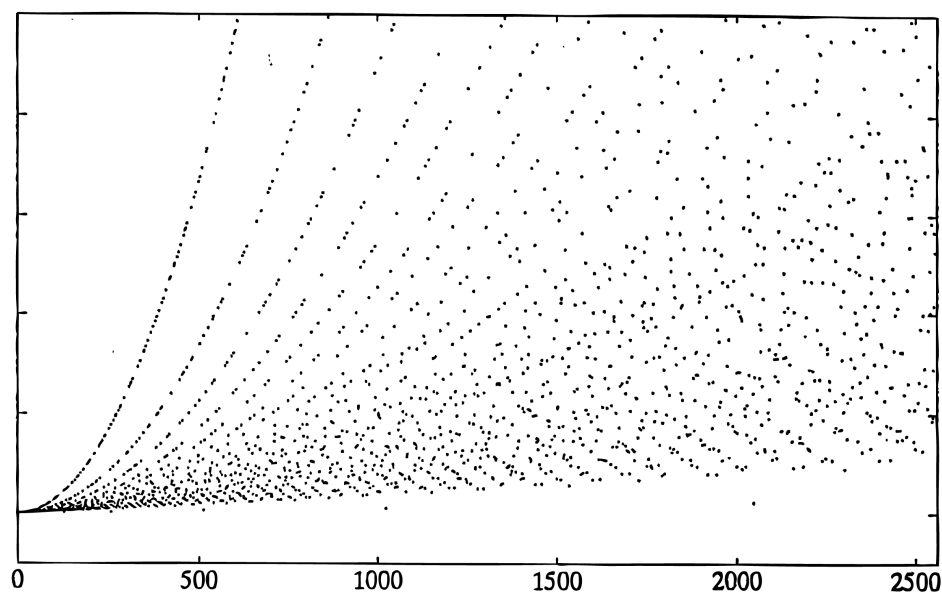
The Glassman-Ferguson FFT is a compact implementation of a mixed-radix Cooley-Tukey FFT with the short DFTs for each factor being calculated by a Goertzel-like algorithm. This means there are twiddle factor multiplications even when the factors are relatively prime, however, the indexing is simple and compact. It will calculate the DFT of a sequence of any length but is efficient only if the length is highly composite. The figures contain plots of the number of floating point multiplications plus additions vs. the length of the FFT. The numbers on the vertical axis have relative meaning but no absolute meaning.



Flop-Count vs Length for the Glassman-Ferguson FFT

Note the parabolic shape of the curve for certain values. The upper curve is for prime lengths, the next one is for lengths that are two times a prime, and

the next one is for lengths that are for three times a prime, etc. The shape of the lower boundary is roughly $N \log N$. The program that generated these two figures used a Cooley-Tukey FFT if the length is two to a power which accounts for the points that are below the major lower boundary.



Flop-Count vs Length for the Glassman-Ferguson FFT

Appendix 3: FFT Computer Programs

Fortran programs for efficient DFT, Cooley-Tukey, and Prime Factor Algorithm FFTs.

Goertzel Algorithm

A FORTRAN implementation of the first-order Goertzel algorithm with in-order input as given in ([link](#)) and [link](#) is given below.

```
C-----
C  GOERTZEL'S  DFT  ALGORITHM
C  First order, input inorder
C  C. S. BURRUS,   SEPT 1983
C-----
      SUBROUTINE DFT(X,Y,A,B,N)
      REAL X(260), Y(260), A(260), B(260)
      Q = 6.283185307179586/N
      DO 20 J=1, N
        C  = COS(Q*(J-1))
        S  = SIN(Q*(J-1))
        AT = X(1)
        BT = Y(1)
        DO 30 I = 2, N
          T  = C*AT - S*BT + X(I)
          BT = C*BT + S*AT + Y(I)
          AT = T
30      CONTINUE
        A(J) = C*AT - S*BT
        B(J) = C*BT + S*AT
20    CONTINUE
      RETURN
      END
```

First Order Goertzel Algorithm

Second Order Goertzel Algorithm

Below is the program for a second order Goertzel algorithm.

```

C-----
C   GOERTZEL'S   DFT   ALGORITHM
C   Second order, input inorder
C   C. S. BURRUS,   SEPT 1983
C-----
      SUBROUTINE DFT(X,Y,A,B,N)
      REAL X(260), Y(260), A(260), B(260)
C
      Q = 6.283185307179586/N
      DO 20 J = 1, N
        C = COS(Q*(J-1))
        S = SIN(Q*(J-1))
        CC = 2*C
        A2 = 0
        B2 = 0
        A1 = X(1)
        B1 = Y(1)
        DO 30 I = 2, N
          T = A1
          A1 = CC*A1 - A2 + X(I)
          A2 = T
          T = B1
          B1 = CC*B1 - B2 + Y(I)
          B2 = T
30      CONTINUE
        A(J) = C*A1 - A2 - S*B1
        B(J) = C*B1 - B2 + S*A1
20    CONTINUE
C
      RETURN
      END

```

Second Order Goertzel Algorithm

Second Order Goertzel Algorithm 2

Second order Goertzel algorithm that calculates two outputs at a time.

```

C-----
-----
C GOERTZEL'S DFT ALGORITHM, Second order
C Input inorder, output by twos; C.S. Burrus,
C SEPT 1991
C-----
-----
      SUBROUTINE DFT(X,Y,A,B,N)
      REAL X(260), Y(260), A(260), B(260)
      Q = 6.283185307179586/N
      DO 20 J = 1, N/2 + 1
          C = COS(Q*(J-1))
          S = SIN(Q*(J-1))
          CC = 2*C
          A2 = 0
          B2 = 0
          A1 = X(1)
          B1 = Y(1)
          DO 30 I = 2, N
              T = A1
              A1 = CC*A1 - A2 + X(I)
              A2 = T
              T = B1
              B1 = CC*B1 - B2 + Y(I)
              B2 = T
30      CONTINUE
          A2 = C*A1 - A2
          T = S*B1
          A(J) = A2 - T
          A(N-J+2) = A2 + T
          B2 = C*B1 - B2
          T = S*A1
          B(J) = B2 + T
          B(N-J+2) = B2 - T
20      CONTINUE
      RETURN
      END

```

Figure. Second Order Goertzel Calculating Two Outputs at a Time

Basic QFT Algorithm

A FORTRAN implementation of the basic QFT algorithm is given below to show how the theory is implemented. The program is written for clarity, not to minimize the number of floating point operations.

C

```
SUBROUTINE QDFT(X,Y,XX,YY,NN)
REAL X(0:260),Y(0:260),XX(0:260),YY(0:260)
C
N1 = NN - 1
N2 = N1/2
N21 = NN/2
Q = 6.283185308/NN
DO 2 K = 0, N21
    SSX = X(0)
    SSY = Y(0)
    SDX = 0
    SDY = 0
    IF (MOD(NN,2).EQ.0) THEN
        SSX = SSX + COS(3.1426*K)*X(N21)
        SSY = SSY + COS(3.1426*K)*Y(N21)
    ENDIF
    DO 3 N = 1, N2
        SSX = SSX + (X(N) + X(NN-N))*COS(Q*N*K)
        SSY = SSY + (Y(N) + Y(NN-N))*COS(Q*N*K)
        SDX = SDX + (X(N) - X(NN-N))*SIN(Q*N*K)
        SDY = SDY + (Y(N) - Y(NN-N))*SIN(Q*N*K)
    3 CONTINUE
    XX(K) = SSX + SDY
    YY(K) = SSY - SDX
    XX(NN-K) = SSX - SDY
    YY(NN-K) = SSY + SDX
```

```

2  CONTINUE
   RETURN
   END

```

Simple QFT Fortran Program

Basic Radix-2 FFT Algorithm

Below is the Fortran code for a simple Decimation-in-Frequency, Radix-2, one butterfly Cooley-Tukey FFT followed by a bit-reversing unscrambler.

```

C
C   A COOLEY-TUKEY RADIX-2, DIF  FFT PROGRAM
C   COMPLEX INPUT DATA IN ARRAYS X AND Y
C       C. S. BURRUS, RICE UNIVERSITY, SEPT 1983
C-----
-----
      SUBROUTINE FFT (X,Y,N,M)
      REAL X(1), Y(1)
C-----MAIN FFT LOOPS-----
-----
C
      N2 = N
      DO 10 K = 1, M
          N1 = N2
          N2 = N2/2
          E = 6.283185307179586/N1
          A = 0
          DO 20 J = 1, N2
              C = COS (A)
              S = SIN (A)
              A = J*E
              DO 30 I = J, N, N1
                  L = I + N2
                  XT = X(I) - X(L)
                  X(I) = X(I) + X(L)
                  YT = Y(I) - Y(L)

```

```

                                Y(I) = Y(I) + Y(L)
                                X(L) = C*XT + S*YT
                                Y(L) = C*YT - S*XT
30                                CONTINUE
20                                CONTINUE
10                                CONTINUE
C
C-----DIGIT REVERSE COUNTER-----
-
100    J = 1
      N1 = N - 1
      DO 104 I=1, N1
          IF (I.GE.J) GOXTO 101
          XT = X(J)
          X(J) = X(I)
          X(I) = XT
          XT = Y(J)
          Y(J) = Y(I)
          Y(I) = XT
101      K = N/2
102      IF (K.GE.J) GOTO 103
          J = J - K
          K = K/2
          GOTO 102
103      J = J + K
104    CONTINUE
      RETURN
      END

```

Figure: Radix-2, DIF, One Butterfly Cooley-Tukey FFT

Basic DIT Radix-2 FFT Algorithm

Below is the Fortran code for a simple Decimation-in-Time, Radix-2, one butterfly Cooley-Tukey FFT preceded by a bit-reversing scrambler.

```

C
C   A COOLEY-TUKEY RADIX-2, DIT  FFT PROGRAM
C   COMPLEX INPUT DATA IN ARRAYS X AND Y
C       C. S. BURRUS, RICE UNIVERSITY, SEPT 1985
C
C-----
-----
      SUBROUTINE FFT (X,Y,N,M)
      REAL X(1), Y(1)
C-----DIGIT REVERSE COUNTER-----
-
C
100    J = 1
      N1 = N - 1
      DO 104 I=1, N1
          IF (I.GE.J) GOTO 101
              XT = X(J)
              X(J) = X(I)
              X(I) = XT
              XT = Y(J)
              Y(J) = Y(I)
              Y(I) = XT
101      K = N/2
102      IF (K.GE.J) GOTO 103
          J = J - K
          K = K/2
          GOTO 102
103      J = J + K
104    CONTINUE
C-----MAIN FFT LOOPS-----
-----
C
      N2 = 1
      DO 10 K = 1, M
          E = 6.283185307179586/(2*N2)
          A = 0
          DO 20 J = 1, N2

```

```

      C = COS (A)
      S = SIN (A)
      A = J*E
      DO 30 I = J, N, 2*N2
          L = I + N2
          XT = C*X(L) + S*Y(L)
          YT = C*Y(L) - S*X(L)
          X(L) = X(I) - XT
          X(I) = X(I) + XT
          Y(L) = Y(I) - YT
          Y(I) = Y(I) + YT
30      CONTINUE
20      CONTINUE
      N2 = N2+N2
10      CONTINUE
C
      RETURN
      END

```

DIF Radix-2 FFT Algorithm

Below is the Fortran code for a Decimation-in-Frequency, Radix-2, three butterfly Cooley-Tukey FFT followed by a bit-reversing unscrambler.

```

C   A COOLEY-TUKEY RADIX 2, DIF  FFT PROGRAM
C   THREE-BF, MULT BY 1  AND  J  ARE REMOVED
C   COMPLEX INPUT DATA IN ARRAYS X AND Y
C   TABLE LOOK-UP OF W VALUES
C       C. S. BURRUS, RICE UNIVERSITY, SEPT 1983
C-----
-----
      SUBROUTINE FFT (X,Y,N,M,WR,WI)
      REAL X(1), Y(1), WR(1), WI(1)
C-----MAIN FFT LOOPS-----
-----
C
      N2 = N

```

```

DO 10 K = 1, M
  N1 = N2
  N2 = N2/2
  JT = N2/2 + 1
  DO 1 I = 1, N, N1
    L = I + N2
    T = X(I) - X(L)
    X(I) = X(I) + X(L)
    X(L) = T
    T = Y(I) - Y(L)
    Y(I) = Y(I) + Y(L)
    Y(L) = T
1    CONTINUE
  IF (K.EQ.M) GOTO 10
  IE = N/N1
  IA = 1
  DO 20 J = 2, N2
    IA = IA + IE
    IF (J.EQ.JT) GOTO 50
    C = WR(IA)
    S = WI(IA)
    DO 30 I = J, N, N1
      L = I + N2
      T = X(I) - X(L)
      X(I) = X(I) + X(L)
      TY = Y(I) - Y(L)
      Y(I) = Y(I) + Y(L)
      X(L) = C*T + S*TY
      Y(L) = C*TY - S*T
30    CONTINUE
    GOTO 25
50    DO 40 I = J, N, N1
      L = I + N2
      T = X(I) - X(L)
      X(I) = X(I) + X(L)
      TY = Y(I) - Y(L)
      Y(I) = Y(I) + Y(L)

```

```

                X(L) = TY
                Y(L) = -T
    40          CONTINUE
    25          A = J*E
    20          CONTINUE
    10  CONTINUE
C-----DIGIT REVERSE COUNTER Goes here-----
-----
    RETURN
    END

```

Basic DIF Radix-4 FFT Algorithm

Below is the Fortran code for a simple Decimation-in-Frequency, Radix-4, one butterfly Cooley-Tukey FFT to be followed by an unscrambler.

```

C  A COOLEY-TUKEY RADIX-4 DIF  FFT PROGRAM
C  COMPLEX INPUT DATA IN ARRAYS X AND Y
C  LENGTH IS  N = 4 ** M
C    C. S. BURRUS, RICE UNIVERSITY, SEPT 1983
C-----
-----
    SUBROUTINE  FFT4 (X,Y,N,M)
    REAL X(1), Y(1)
C-----MAIN FFT LOOPS-----
-----
    N2 = N
    DO 10 K = 1, M
        N1 = N2
        N2 = N2/4
        E = 6.283185307179586/N1
        A = 0
C-----MAIN BUTTERFLIES-----
-----
        DO 20 J=1, N2
            B      = A + A

```

```

C      = A + B
C01    = COS(A)
C02    = COS(B)
C03    = COS(C)
SI1    = SIN(A)
SI2    = SIN(B)
SI3    = SIN(C)
A      = J*E

```

```

C-----BUTTERFLIES WITH SAME W-----
-----

```

```

DO 30 I=J, N, N1
I1 = I  + N2
I2 = I1 + N2
I3 = I2 + N2
R1 = X(I ) + X(I2)
R3 = X(I ) - X(I2)
S1 = Y(I ) + Y(I2)
S3 = Y(I ) - Y(I2)
R2 = X(I1) + X(I3)
R4 = X(I1) - X(I3)
S2 = Y(I1) + Y(I3)
S4 = Y(I1) - Y(I3)
X(I) = R1 + R2
R2    = R1 - R2
R1    = R3 - S4
R3    = R3 + S4
Y(I)  = S1 + S2
S2    = S1 - S2
S1    = S3 + R4
S3    = S3 - R4
X(I1) = C01*R3 + SI1*S3
Y(I1) = C01*S3 - SI1*R3
X(I2) = C02*R2 + SI2*S2
Y(I2) = C02*S2 - SI2*R2
X(I3) = C03*R1 + SI3*S1
Y(I3) = C03*S1 - SI3*R1

```

```

20          CONTINUE
10          CONTINUE
C-----DIGIT REVERSE COUNTER goes here-----
      RETURN
      END

```

Basic DIF Radix-4 FFT Algorithm

Below is the Fortran code for a Decimation-in-Frequency, Radix-4, three butterfly Cooley-Tukey FFT followed by a bit-reversing unscrambler. Twiddle factors are precalculated and stored in arrays WR and WI.

```

C
C   A COOLEY-TUKEY RADIX-4 DIF  FFT PROGRAM
C   THREE BF, MULTIPLICATIONS BY 1, J, ETC. ARE
C   REMOVED
C   COMPLEX INPUT DATA IN ARRAYS X AND Y
C   LENGTH IS  N = 4 ** M
C   TABLE LOOKUP OF W VALUES
C
C       C. S. BURRUS, RICE UNIVERSITY,  SEPT 1983
C
C-----
C
C       SUBROUTINE  FFT4 (X,Y,N,M,WR,WI)
C       REAL X(1), Y(1), WR(1), WI(1)
C       DATA C21 / 0.707106778 /
C
C-----MAIN FFT LOOPS-----
C-----
C
C       N2 = N
C       DO 10 K = 1, M
C           N1 = N2
C           N2 = N2/4
C           JT = N2/2 + 1

```

C-----SPECIAL BUTTERFLY FOR W = 1-----

```
DO 1 I = 1, N, N1
  I1 = I  + N2
  I2 = I1 + N2
  I3 = I2 + N2
  R1 = X(I ) + X(I2)
  R3 = X(I ) - X(I2)
  S1 = Y(I ) + Y(I2)
  S3 = Y(I ) - Y(I2)
  R2 = X(I1) + X(I3)
  R4 = X(I1) - X(I3)
  S2 = Y(I1) + Y(I3)
  S4 = Y(I1) - Y(I3)
```

C

```
  X(I) = R1 + R2
  X(I2)= R1 - R2
  X(I3)= R3 - S4
  X(I1)= R3 + S4
```

C

```
  Y(I) = S1 + S2
  Y(I2)= S1 - S2
  Y(I3)= S3 + R4
  Y(I1)= S3 - R4
```

C

```
1      CONTINUE
      IF (K.EQ.M) GOTO 10
      IE = N/N1
      IA1 = 1
```

C-----GENERAL BUTTERFLY-----

```
DO 20 J = 2, N2
  IA1 = IA1 + IE
  IF (J.EQ.JT) GOTO 50
  IA2 = IA1 + IA1 - 1
  IA3 = IA2 + IA1 - 1
  C01 = WR(IA1)
```

```

C02  = WR(IA2)
C03  = WR(IA3)
SI1  = WI(IA1)
SI2  = WI(IA2)
SI3  = WI(IA3)

```

C-----BUTTERFLIES WITH SAME W-----

```

DO 30 I = J, N, N1
I1 = I  + N2
I2 = I1 + N2
I3 = I2 + N2
R1 = X(I ) + X(I2)
R3 = X(I ) - X(I2)
S1 = Y(I ) + Y(I2)
S3 = Y(I ) - Y(I2)
R2 = X(I1) + X(I3)
R4 = X(I1) - X(I3)
S2 = Y(I1) + Y(I3)
S4 = Y(I1) - Y(I3)

```

C

```

X(I) = R1 + R2
R2    = R1 - R2
R1    = R3 - S4
R3    = R3 + S4

```

C

```

Y(I) = S1 + S2
S2    = S1 - S2
S1    = S3 + R4
S3    = S3 - R4

```

C

```

X(I1) = C01*R3 + SI1*S3
Y(I1) = C01*S3 - SI1*R3
X(I2) = C02*R2 + SI2*S2
Y(I2) = C02*S2 - SI2*R2
X(I3) = C03*R1 + SI3*S1
Y(I3) = C03*S1 - SI3*R1

```

```

                GOTO 20
C-----SPECIAL BUTTERFLY FOR  W = J---
-----
50              DO 40 I = J, N, N1
                I1 = I  + N2
                I2 = I1 + N2
                I3 = I2 + N2
                R1 = X(I ) + X(I2)
                R3 = X(I ) - X(I2)
                S1 = Y(I ) + Y(I2)
                S3 = Y(I ) - Y(I2)
                R2 = X(I1) + X(I3)
                R4 = X(I1) - X(I3)
                S2 = Y(I1) + Y(I3)
                S4 = Y(I1) - Y(I3)
C
                X(I) = R1 + R2
                Y(I2) = -R1 + R2
                R1     = R3 - S4
                R3     = R3 + S4
C
                Y(I) = S1 + S2
                X(I2) = S1 - S2
                S1     = S3 + R4
                S3     = S3 - R4
C
                X(I1) = (S3 + R3)*C21
                Y(I1) = (S3 - R3)*C21
                X(I3) = (S1 - R1)*C21
                Y(I3) = -(S1 + R1)*C21
40              CONTINUE
20              CONTINUE
10              CONTINUE
C-----DIGIT REVERSE COUNTER-----
100            J = 1
                N1 = N - 1
                DO 104 I = 1, N1

```

```

        IF (I.GE.J) GOTO 101
        R1    = X(J)
        X(J)  = X(I)
        X(I)  = R1
        R1    = Y(J)
        Y(J)  = Y(I)
        Y(I)  = R1
101      K = N/4
102      IF (K*3.GE.J) GOTO 103
           J = J - K*3
           K = K/4
           GOTO 102
103      J = J + K
104      CONTINUE
        RETURN
        END

```

Basic DIF Split Radix FFT Algorithm

Below is the Fortran code for a simple Decimation-in-Frequency, Split-Radix, one butterfly FFT to be followed by a bit-reversing unscrambler.

```

C    A DUHAMEL-HOLLMANN SPLIT RADIX  FFT PROGRAM
C    FROM: ELECTRONICS LETTERS, JAN. 5, 1984
C    COMPLEX INPUT DATA IN ARRAYS X AND Y
C    LENGTH IS  N = 2 ** M
C      C. S. BURRUS, RICE UNIVERSITY, MARCH 1984
C
C-----
C-----
      SUBROUTINE  FFT (X,Y,N,M)
      REAL X(1), Y(1)
C-----MAIN FFT LOOPS-----
C-----
C
      N1 = N

```

```

N2 = N/2
IP = 0
IS = 1
A = 6.283185307179586/N
DO 10 K = 1, M-1
    JD = N1 + N2
    N1 = N2
    N2 = N2/2
    J0 = N1*IP + 1
    IP = 1 - IP
    DO 20 J = J0, N, JD
        JS = 0
        JT = J + N2 - 1
        DO 30 I = J, JT
            JSS= JS*IS
            JS = JS + 1
            C1 = COS(A*JSS)
            C3 = COS(3*A*JSS)
            S1 = -SIN(A*JSS)
            S3 = -SIN(3*A*JSS)
            I1 = I + N2
            I2 = I1 + N2
            I3 = I2 + N2
            R1 = X(I ) + X(I2)
            R2 = X(I ) - X(I2)
            R3 = X(I1) - X(I3)
            X(I2) = X(I1) + X(I3)
            X(I1) = R1

```

C

```

R1 = Y(I ) + Y(I2)
R4 = Y(I ) - Y(I2)
R5 = Y(I1) - Y(I3)
Y(I2) = Y(I1) + Y(I3)
Y(I1) = R1

```

C

```

R1 = R2 - R5
R2 = R2 + R5

```

```

          R5      = R4 + R3
          R4      = R4 - R3
C
          X(I)    = C1*R1 + S1*R5
          Y(I)    = C1*R5 - S1*R1
          X(I3)   = C3*R2 + S3*R4
          Y(I3)   = C3*R4 - S3*R2
30      CONTINUE
20      CONTINUE
        IS = IS + IS
10      CONTINUE
        IP = 1 - IP
        J0 = 2 - IP
        DO 5 I = J0, N-1, 3
            I1 = I + 1
            R1   = X(I) + X(I1)
            X(I1) = X(I) - X(I1)
            X(I)  = R1
            R1   = Y(I) + Y(I1)
            Y(I1) = Y(I) - Y(I1)
            Y(I)  = R1
5      CONTINUE
        RETURN
        END

```

DIF Split Radix FFT Algorithm

Below is the Fortran code for a simple Decimation-in-Frequency, Split-Radix, two butterfly FFT to be followed by a bit-reversing unscrambler. Twiddle factors are precalculated and stored in arrays WR and WI.

```

C-----
-----C
C      A DUHAMEL-HOLLMAN SPLIT RADIX FFT
C
C      REF: ELECTRONICS LETTERS, JAN. 5, 1984
C

```

```

C      COMPLEX INPUT AND OUTPUT DATA IN ARRAYS X
AND Y      C
C      LENGTH IS  $N = 2^{**} M$ , OUTPUT IN BIT-
REVERSED ORDER      C
C      TWO BUTTERFLIES TO REMOVE MULTS BY UNITY
C
C      SPECIAL LAST TWO STAGES
C
C      TABLE LOOK-UP OF SINE AND COSINE VALUES
C
C      C.S. BURRUS,      RICE UNIV.      APRIL
1985      C
C-----
-----C

```

```

C
      SUBROUTINE FFT(X,Y,N,M,WR,WI)
      REAL X(1),Y(1),WR(1),WI(1)
      C81= 0.707106778
      N2 = 2*N
      DO 10 K = 1, M-3
      IS = 1
      ID = N2
      N2 = N2/2
      N4 = N2/4
2      DO 1 I0 = IS, N-1, ID
      I1 = I0 + N4
      I2 = I1 + N4
      I3 = I2 + N4
      R1 = X(I0) - X(I2)
      X(I0) = X(I0) + X(I2)
      R2 = Y(I1) - Y(I3)
      Y(I1) = Y(I1) + Y(I3)
      X(I2) = R1 + R2
      R2 = R1 - R2
      R1 = X(I1) - X(I3)
      X(I1) = X(I1) + X(I3)
      X(I3) = R2

```

```

        R2      = Y(I0) - Y(I2)
        Y(I0)   = Y(I0) + Y(I2)
        Y(I2)   = -R1 + R2
        Y(I3)   = R1 + R2
1      CONTINUE
        IS = 2*ID - N2 + 1
        ID = 4*ID
        IF (IS.LT.N) GOTO 2
        IE   = N/N2
        IA1  = 1
        DO 20 J = 2, N4
            IA1 = IA1 + IE
            IA3 = 3*IA1 - 2
            CC1 = WR(IA1)
            SS1 = WI(IA1)
            CC3 = WR(IA3)
            SS3 = WI(IA3)
            IS  = J
            ID  = 2*N2
40      DO 30 I0 = IS, N-1, ID
            I1 = I0 + N4
            I2 = I1 + N4
            I3 = I2 + N4

C
            R1      = X(I0) - X(I2)
            X(I0)   = X(I0) + X(I2)
            R2      = X(I1) - X(I3)
            X(I1)   = X(I1) + X(I3)
            S1      = Y(I0) - Y(I2)
            Y(I0)   = Y(I0) + Y(I2)
            S2      = Y(I1) - Y(I3)
            Y(I1)   = Y(I1) + Y(I3)

C
            S3      = R1 - S2
            R1      = R1 + S2
            S2      = R2 - S1
            R2      = R2 + S1

```

```

X(I2) = R1*CC1 - S2*SS1
Y(I2) = -S2*CC1 - R1*SS1
X(I3) = S3*CC3 + R2*SS3
Y(I3) = R2*CC3 - S3*SS3
30      CONTINUE
      IS = 2*ID - N2 + J
      ID = 4*ID
      IF (IS.LT.N) GOTO 40
20      CONTINUE
10      CONTINUE
C
      IS = 1
      ID = 32
50      DO 60 I = IS, N, ID
          I0 = I + 8
          DO 15 J = 1, 2
              R1 = X(I0) + X(I0+2)
              R3 = X(I0) - X(I0+2)
              R2 = X(I0+1) + X(I0+3)
              R4 = X(I0+1) - X(I0+3)
              X(I0) = R1 + R2
              X(I0+1) = R1 - R2
              R1 = Y(I0) + Y(I0+2)
              S3 = Y(I0) - Y(I0+2)
              R2 = Y(I0+1) + Y(I0+3)
              S4 = Y(I0+1) - Y(I0+3)
              Y(I0) = R1 + R2
              Y(I0+1) = R1 - R2
              Y(I0+2) = S3 - R4
              Y(I0+3) = S3 + R4
              X(I0+2) = R3 + S4
              X(I0+3) = R3 - S4
              I0 = I0 + 4
15      CONTINUE
60      CONTINUE
      IS = 2*ID - 15
      ID = 4*ID

```

IF (IS.LT.N) GOTO 50

C

IS = 1

ID = 16

55 DO 65 I0 = IS, N, ID

R1 = X(I0) + X(I0+4)

R5 = X(I0) - X(I0+4)

R2 = X(I0+1) + X(I0+5)

R6 = X(I0+1) - X(I0+5)

R3 = X(I0+2) + X(I0+6)

R7 = X(I0+2) - X(I0+6)

R4 = X(I0+3) + X(I0+7)

R8 = X(I0+3) - X(I0+7)

T1 = R1 - R3

R1 = R1 + R3

R3 = R2 - R4

R2 = R2 + R4

X(I0) = R1 + R2

X(I0+1) = R1 - R2

C

R1 = Y(I0) + Y(I0+4)

S5 = Y(I0) - Y(I0+4)

R2 = Y(I0+1) + Y(I0+5)

S6 = Y(I0+1) - Y(I0+5)

S3 = Y(I0+2) + Y(I0+6)

S7 = Y(I0+2) - Y(I0+6)

R4 = Y(I0+3) + Y(I0+7)

S8 = Y(I0+3) - Y(I0+7)

T2 = R1 - S3

R1 = R1 + S3

S3 = R2 - R4

R2 = R2 + R4

Y(I0) = R1 + R2

Y(I0+1) = R1 - R2

X(I0+2) = T1 + S3

X(I0+3) = T1 - S3

Y(I0+2) = T2 - R3

```

C          Y(I0+3) = T2 + R3

C          R1 = (R6 - R8)*C81
          R6 = (R6 + R8)*C81
          R2 = (S6 - S8)*C81
          S6 = (S6 + S8)*C81

C          T1 = R5 - R1
          R5 = R5 + R1
          R8 = R7 - R6
          R7 = R7 + R6
          T2 = S5 - R2
          S5 = S5 + R2
          S8 = S7 - S6
          S7 = S7 + S6
          X(I0+4) = R5 + S7
          X(I0+7) = R5 - S7
          X(I0+5) = T1 + S8
          X(I0+6) = T1 - S8
          Y(I0+4) = S5 - R7
          Y(I0+7) = S5 + R7
          Y(I0+5) = T2 - R8
          Y(I0+6) = T2 + R8

65      CONTINUE
          IS = 2*ID - 7
          ID = 4*ID
          IF (IS.LT.N) GOTO 55

C
C-----BIT REVERSE COUNTER-----
C
100      J = 1
          N1 = N - 1
          DO 104 I=1, N1
              IF (I.GE.J) GOTO 101
              XT = X(J)
              X(J) = X(I)
              X(I) = XT
          101
          J = J + 1
          104

```

```

          XT    = Y(J)
          Y(J)  = Y(I)
          Y(I)  = XT
101      K = N/2
102      IF (K.GE.J) GOTO 103
          J = J - K
          K = K/2
          GOTO 102
103      J = J + K
104      CONTINUE
          RETURN
          END

```

Prime Factor FFT Algorithm

Below is the Fortran code for a Prime-Factor Algorithm (PFA) FFT allowing factors of the length of 2, 3, 4, 5, and 7. It is followed by an unscrambler.

```

C-----
C
C
C   A PRIME FACTOR FFT PROGRAM WITH GENERAL
MODULES
C   COMPLEX INPUT DATA IN ARRAYS  X AND Y
C   COMPLEX OUTPUT IN  A AND B
C   LENGTH  N  WITH  M  FACTORS IN ARRAY  NI
C   N = NI(1)*NI(2)* ... *NI(M)
C   UNSCRAMBLING CONSTANT  UNSC
C   UNSC = N/NI(1) + N/NI(2) +...+ N/NI(M), MOD
N
C   C. S. BURRUS, RICE UNIVERSITY, JAN 1987
C
C-----
C
C
SUBROUTINE PFA(X,Y,N,M,NI,A,B,UNSC)

```

C

```
INTEGER  NI(4), I(16), UNSC
        REAL X(1), Y(1), A(1), B(1)
```

C

```
DATA  C31, C32  / -0.86602540, -1.50000000 /
DATA  C51, C52  /  0.95105652, -1.53884180 /
DATA  C53, C54  / -0.36327126,  0.55901699 /
DATA  C55       / -1.25      /
DATA  C71, C72  / -1.16666667, -0.79015647 /
DATA  C73, C74  /  0.055854267,  0.7343022 /
DATA  C75, C76  /  0.44095855, -0.34087293 /
DATA  C77, C78  /  0.53396936,  0.87484229 /
```

C

C-----NESTED LOOPS-----
--

C

```
DO 10 K=1, M
    N1 = NI(K)
    N2 = N/N1
    DO 15 J=1, N, N1
        IT = J
        DO 30 L=1, N1
            I(L) = IT
            A(L) = X(IT)
            B(L) = Y(IT)
            IT = IT + N2
            IF (IT.GT.N) IT = IT - N
30      CONTINUE
        GOTO (20,102,103,104,105,20,107), N1
```

C

C-----WFTA N=2-----

C

```
102    R1      = A(1)
      A(1)     = R1 + A(2)
      A(2)     = R1 - A(2)
```

C

$R1 = B(1)$
 $B(1) = R1 + B(2)$
 $B(2) = R1 - B(2)$

C

GOTO 20

C-----WFTA N=3-----

C

103 $R2 = (A(2) - A(3)) * C31$
 $R1 = A(2) + A(3)$
 $A(1) = A(1) + R1$
 $R1 = A(1) + R1 * C32$

C

$S2 = (B(2) - B(3)) * C31$
 $S1 = B(2) + B(3)$
 $B(1) = B(1) + S1$
 $S1 = B(1) + S1 * C32$

C

$A(2) = R1 - S2$
 $A(3) = R1 + S2$
 $B(2) = S1 + R2$
 $B(3) = S1 - R2$

C

GOTO 20

C

C-----WFTA N=4-----

C

104 $R1 = A(1) + A(3)$
 $T1 = A(1) - A(3)$
 $R2 = A(2) + A(4)$
 $A(1) = R1 + R2$
 $A(3) = R1 - R2$

C

$R1 = B(1) + B(3)$
 $T2 = B(1) - B(3)$
 $R2 = B(2) + B(4)$

$B(1) = R1 + R2$
 $B(3) = R1 - R2$

C

$R1 = A(2) - A(4)$
 $R2 = B(2) - B(4)$

C

$A(2) = T1 + R2$
 $A(4) = T1 - R2$
 $B(2) = T2 - R1$
 $B(4) = T2 + R1$

C

GOTO 20

C

C-----WFTA N=5-----

C

105 $R1 = A(2) + A(5)$
 $R4 = A(2) - A(5)$
 $R3 = A(3) + A(4)$
 $R2 = A(3) - A(4)$

C

$T = (R1 - R3) * C54$
 $R1 = R1 + R3$
 $A(1) = A(1) + R1$
 $R1 = A(1) + R1 * C55$

C

$R3 = R1 - T$
 $R1 = R1 + T$

C

$T = (R4 + R2) * C51$
 $R4 = T + R4 * C52$
 $R2 = T + R2 * C53$

C

$S1 = B(2) + B(5)$
 $S4 = B(2) - B(5)$
 $S3 = B(3) + B(4)$
 $S2 = B(3) - B(4)$

C

```
T = (S1 - S3) * C54
S1 = S1 + S3
B(1) = B(1) + S1
S1 = B(1) + S1 * C55
```

C

```
S3 = S1 - T
S1 = S1 + T
```

C

```
T = (S4 + S2) * C51
S4 = T + S4 * C52
S2 = T + S2 * C53
```

C

```
A(2) = R1 + S2
A(5) = R1 - S2
A(3) = R3 - S4
A(4) = R3 + S4
```

C

```
B(2) = S1 - R2
B(5) = S1 + R2
B(3) = S3 + R4
B(4) = S3 - R4
```

C

```
GOTO 20
```

C-----WFTA N=7-----

--

C

```
107 R1 = A(2) + A(7)
R6 = A(2) - A(7)
S1 = B(2) + B(7)
S6 = B(2) - B(7)
R2 = A(3) + A(6)
R5 = A(3) - A(6)
S2 = B(3) + B(6)
S5 = B(3) - B(6)
R3 = A(4) + A(5)
R4 = A(4) - A(5)
```

$$S3 = B(4) + B(5)$$

$$S4 = B(4) - B(5)$$

C

$$T3 = (R1 - R2) * C74$$

$$T = (R1 - R3) * C72$$

$$R1 = R1 + R2 + R3$$

$$A(1) = A(1) + R1$$

$$R1 = A(1) + R1 * C71$$

$$R2 = (R3 - R2) * C73$$

$$R3 = R1 - T + R2$$

$$R2 = R1 - R2 - T3$$

$$R1 = R1 + T + T3$$

$$T = (R6 - R5) * C78$$

$$T3 = (R6 + R4) * C76$$

$$R6 = (R6 + R5 - R4) * C75$$

$$R5 = (R5 + R4) * C77$$

$$R4 = R6 - T3 + R5$$

$$R5 = R6 - R5 - T$$

$$R6 = R6 + T3 + T$$

C

$$T3 = (S1 - S2) * C74$$

$$T = (S1 - S3) * C72$$

$$S1 = S1 + S2 + S3$$

$$B(1) = B(1) + S1$$

$$S1 = B(1) + S1 * C71$$

$$S2 = (S3 - S2) * C73$$

$$S3 = S1 - T + S2$$

$$S2 = S1 - S2 - T3$$

$$S1 = S1 + T + T3$$

$$T = (S6 - S5) * C78$$

$$T3 = (S6 + S4) * C76$$

$$S6 = (S6 + S5 - S4) * C75$$

$$S5 = (S5 + S4) * C77$$

$$S4 = S6 - T3 + S5$$

$$S5 = S6 - S5 - T$$

$$S6 = S6 + T3 + T$$

C

```

A(2) = R3 + S4
A(7) = R3 - S4
A(3) = R1 + S6
A(6) = R1 - S6
A(4) = R2 - S5
A(5) = R2 + S5
B(4) = S2 + R5
B(5) = S2 - R5
B(2) = S3 - R4
B(7) = S3 + R4
B(3) = S1 - R6
B(6) = S1 + R6

```

C

```

20          IT    = J
           DO 31 L=1, N1
               I(L) = IT
           X(IT) = A(L)
           Y(IT) = B(L)
               IT = IT + N2
               IF (IT.GT.N) IT = IT - N
31          CONTINUE
15          CONTINUE
10          CONTINUE

```

C

C-----UNSCRAMBLING-----

--

C

```

L = 1
DO 2 K=1, N
    A(K) = X(L)
    B(K) = Y(L)
    L = L + UNSC
    IF (L.GT.N) L = L - N
2    CONTINUE
RETURN
END

```

C

In Place, In Order Prime Factor FFT Algorithm

Below is the Fortran code for a Prime-Factor Algorithm (PFA) FFT allowing factors of the length of 2, 3, 4, 5, 7, 8, 9, and 16. It is both in-place and in-order, so requires no unscrambler.

```
C
C   A PRIME FACTOR FFT PROGRAM
C   IN-PLACE AND IN-ORDER
C   COMPLEX INPUT DATA IN ARRAYS  X AND Y
C   LENGTH  N  WITH  M  FACTORS IN ARRAY  NI
C       N = NI(1)*NI(2)*...*NI(M)
C   REDUCED TEMP STORAGE IN SHORT WFTA MODULES
C   Has modules 2,3,4,5,7,8,9,16
C   PROGRAM BY  C. S. BURRUS,  RICE UNIVERSITY
C               SEPT 1983
C-----
---
C
  SUBROUTINE PFA(X,Y,N,M,NI)
    INTEGER  NI(4), I(16), IP(16), LP(16)
    REAL X(1), Y(1)
    DATA  C31, C32  / -0.86602540, -1.50000000 /
    DATA  C51, C52  /  0.95105652, -1.53884180 /
    DATA  C53, C54  / -0.36327126,  0.55901699 /
    DATA  C55       / -1.25      /
    DATA  C71, C72  / -1.16666667, -0.79015647 /
    DATA  C73, C74  /  0.055854267,  0.7343022 /
    DATA  C75, C76  /  0.44095855, -0.34087293 /
    DATA  C77, C78  /  0.53396936,  0.87484229 /
    DATA  C81       /  0.70710678 /
    DATA  C95       / -0.50000000 /
    DATA  C92, C93  /  0.93969262, -0.17364818 /
    DATA  C94, C96  /  0.76604444, -0.34202014 /
    DATA  C97, C98  / -0.98480775, -0.64278761 /
    DATA  C162, C163 /  0.38268343,  1.30656297 /
    DATA  C164, C165 /  0.54119610,  0.92387953 /
```

```

C
C-----NESTED LOOPS-----
-----

```

```

C
    DO 10 K=1, M
        N1 = NI(K)
        N2 = N/N1
        L = 1
        N3 = N2 - N1*(N2/N1)
        DO 15 J = 1, N1
            LP(J) = L
            L = L + N3
            IF (L.GT.N1) L = L - N1
15      CONTINUE

```

```

C
    DO 20 J=1, N, N1
        IT = J
        DO 30 L=1, N1
            I(L) = IT
            IP(LP(L)) = IT
            IT = IT + N2
            IF (IT.GT.N) IT = IT - N
30      CONTINUE
        GOTO (20,102,103,104,105,20,107,108,109,
+           20,20,20,20,20,20,116),N1

```

```

C-----WFTA N=2-----
-----

```

```

C
102   R1      = X(I(1))
      X(I(1)) = R1 + X(I(2))
      X(I(2)) = R1 - X(I(2))

```

```

C
      R1      = Y(I(1))
      Y(IP(1)) = R1 + Y(I(2))
      Y(IP(2)) = R1 - Y(I(2))

```

```

C

```

```

      GOTO 20
C
C-----WFTA N=3-----
-----
C
103   R2 = (X(I(2)) - X(I(3))) * C31
      R1 = X(I(2)) + X(I(3))
      X(I(1))= X(I(1)) + R1
      R1      = X(I(1)) + R1 * C32
C
      S2 = (Y(I(2)) - Y(I(3))) * C31
      S1 = Y(I(2)) + Y(I(3))
      Y(I(1))= Y(I(1)) + S1
      S1      = Y(I(1)) + S1 * C32
C
      X(IP(2)) = R1 - S2
      X(IP(3)) = R1 + S2
      Y(IP(2)) = S1 + R2
      Y(IP(3)) = S1 - R2
C
      GOTO 20
C
C-----WFTA N=4-----
-----
C
104   R1 = X(I(1)) + X(I(3))
      T1 = X(I(1)) - X(I(3))
      R2 = X(I(2)) + X(I(4))
      X(IP(1)) = R1 + R2
      X(IP(3)) = R1 - R2
C
      R1 = Y(I(1)) + Y(I(3))
      T2 = Y(I(1)) - Y(I(3))
      R2 = Y(I(2)) + Y(I(4))
      Y(IP(1)) = R1 + R2
      Y(IP(3)) = R1 - R2
C

```

R1 = X(I(2)) - X(I(4))
R2 = Y(I(2)) - Y(I(4))

C

X(IP(2)) = T1 + R2
X(IP(4)) = T1 - R2
Y(IP(2)) = T2 - R1
Y(IP(4)) = T2 + R1

C

GOTO 20

C-----WFTA N=5-----

C

105 R1 = X(I(2)) + X(I(5))
R4 = X(I(2)) - X(I(5))
R3 = X(I(3)) + X(I(4))
R2 = X(I(3)) - X(I(4))

C

T = (R1 - R3) * C54
R1 = R1 + R3
X(I(1)) = X(I(1)) + R1
R1 = X(I(1)) + R1 * C55

C

R3 = R1 - T
R1 = R1 + T

C

T = (R4 + R2) * C51
R4 = T + R4 * C52
R2 = T + R2 * C53

C

S1 = Y(I(2)) + Y(I(5))
S4 = Y(I(2)) - Y(I(5))
S3 = Y(I(3)) + Y(I(4))
S2 = Y(I(3)) - Y(I(4))

C

T = (S1 - S3) * C54
S1 = S1 + S3

$Y(I(1)) = Y(I(1)) + S1$
 $S1 = Y(I(1)) + S1 * C55$

C

$S3 = S1 - T$
 $S1 = S1 + T$

C

$T = (S4 + S2) * C51$
 $S4 = T + S4 * C52$
 $S2 = T + S2 * C53$

C

$X(IP(2)) = R1 + S2$
 $X(IP(5)) = R1 - S2$
 $X(IP(3)) = R3 - S4$
 $X(IP(4)) = R3 + S4$

C

$Y(IP(2)) = S1 - R2$
 $Y(IP(5)) = S1 + R2$
 $Y(IP(3)) = S3 + R4$
 $Y(IP(4)) = S3 - R4$

C

GOTO 20

C-----WFTA N=7-----

--

C

107 $R1 = X(I(2)) + X(I(7))$
 $R6 = X(I(2)) - X(I(7))$
 $S1 = Y(I(2)) + Y(I(7))$
 $S6 = Y(I(2)) - Y(I(7))$
 $R2 = X(I(3)) + X(I(6))$
 $R5 = X(I(3)) - X(I(6))$
 $S2 = Y(I(3)) + Y(I(6))$
 $S5 = Y(I(3)) - Y(I(6))$
 $R3 = X(I(4)) + X(I(5))$
 $R4 = X(I(4)) - X(I(5))$
 $S3 = Y(I(4)) + Y(I(5))$
 $S4 = Y(I(4)) - Y(I(5))$

C

```
T3 = (R1 - R2) * C74
T  = (R1 - R3) * C72
R1 = R1 + R2 + R3
X(I(1)) = X(I(1)) + R1
R1      = X(I(1)) + R1 * C71
R2 =(R3 - R2) * C73
R3 = R1 - T + R2
R2 = R1 - R2 - T3
R1 = R1 + T + T3
T  = (R6 - R5) * C78
T3 =(R6 + R4) * C76
R6 =(R6 + R5 - R4) * C75
R5 =(R5 + R4) * C77
R4 = R6 - T3 + R5
R5 = R6 - R5 - T
R6 = R6 + T3 + T
```

C

```
T3 = (S1 - S2) * C74
T  = (S1 - S3) * C72
S1 = S1 + S2 + S3
Y(I(1)) = Y(I(1)) + S1
S1      = Y(I(1)) + S1 * C71
S2 =(S3 - S2) * C73
S3 = S1 - T  + S2
S2 = S1 - S2 - T3
S1 = S1 + T  + T3
T  = (S6 - S5) * C78
T3 = (S6 + S4) * C76
S6 = (S6 + S5 - S4) * C75
S5 = (S5 + S4) * C77
S4 = S6 - T3 + S5
S5 = S6 - S5 - T
S6 = S6 + T3 + T
```

C

```
X(IP(2)) = R3 + S4
X(IP(7)) = R3 - S4
```

```

X(IP(3)) = R1 + S6
X(IP(6)) = R1 - S6
X(IP(4)) = R2 - S5
X(IP(5)) = R2 + S5
Y(IP(4)) = S2 + R5
Y(IP(5)) = S2 - R5
Y(IP(2)) = S3 - R4
Y(IP(7)) = S3 + R4
Y(IP(3)) = S1 - R6
Y(IP(6)) = S1 + R6

```

C

```
GOTO 20
```

C-----WFTA N=8-----
--

C

```

108   R1 = X(I(1)) + X(I(5))
      R2 = X(I(1)) - X(I(5))
      R3 = X(I(2)) + X(I(8))
      R4 = X(I(2)) - X(I(8))
      R5 = X(I(3)) + X(I(7))
      R6 = X(I(3)) - X(I(7))
      R7 = X(I(4)) + X(I(6))
      R8 = X(I(4)) - X(I(6))
      T1 = R1 + R5
      T2 = R1 - R5
      T3 = R3 + R7
      R3 =(R3 - R7) * C81
      X(IP(1)) = T1 + T3
      X(IP(5)) = T1 - T3
      T1 = R2 + R3
      T3 = R2 - R3
      S1 = R4 - R8
      R4 =(R4 + R8) * C81
      S2 = R4 + R6
      S3 = R4 - R6
      R1 = Y(I(1)) + Y(I(5))

```

```

R2 = Y(I(1)) - Y(I(5))
R3 = Y(I(2)) + Y(I(8))
R4 = Y(I(2)) - Y(I(8))
R5 = Y(I(3)) + Y(I(7))
R6 = Y(I(3)) - Y(I(7))
R7 = Y(I(4)) + Y(I(6))
R8 = Y(I(4)) - Y(I(6))
T4 = R1 + R5
R1 = R1 - R5
R5 = R3 + R7
R3 =(R3 - R7) * C81
Y(IP(1)) = T4 + R5
Y(IP(5)) = T4 - R5
R5 = R2 + R3
R2 = R2 - R3
R3 = R4 - R8
R4 =(R4 + R8) * C81
R7 = R4 + R6
R4 = R4 - R6
X(IP(2)) = T1 + R7
X(IP(8)) = T1 - R7
X(IP(3)) = T2 + R3
X(IP(7)) = T2 - R3
X(IP(4)) = T3 + R4
X(IP(6)) = T3 - R4
Y(IP(2)) = R5 - S2
Y(IP(8)) = R5 + S2
Y(IP(3)) = R1 - S1
Y(IP(7)) = R1 + S1
Y(IP(4)) = R2 - S3
Y(IP(6)) = R2 + S3

```

C

```

GOTO 20

```

C-----WFTA N=9-----

C

```

109    R1 = X(I(2)) + X(I(9))

```

$$\begin{aligned}
R2 &= X(I(2)) - X(I(9)) \\
R3 &= X(I(3)) + X(I(8)) \\
R4 &= X(I(3)) - X(I(8)) \\
R5 &= X(I(4)) + X(I(7)) \\
T8 &= (X(I(4)) - X(I(7))) * C31 \\
R7 &= X(I(5)) + X(I(6)) \\
R8 &= X(I(5)) - X(I(6)) \\
T0 &= X(I(1)) + R5 \\
T7 &= X(I(1)) + R5 * C95 \\
R5 &= R1 + R3 + R7 \\
X(I(1)) &= T0 + R5 \\
T5 &= T0 + R5 * C95 \\
T3 &= (R3 - R7) * C92 \\
R7 &= (R1 - R7) * C93 \\
R3 &= (R1 - R3) * C94 \\
T1 &= T7 + T3 + R3 \\
T3 &= T7 - T3 - R7 \\
T7 &= T7 + R7 - R3 \\
T6 &= (R2 - R4 + R8) * C31 \\
T4 &= (R4 + R8) * C96 \\
R8 &= (R2 - R8) * C97 \\
R2 &= (R2 + R4) * C98 \\
T2 &= T8 + T4 + R2 \\
T4 &= T8 - T4 - R8 \\
T8 &= T8 + R8 - R2
\end{aligned}$$

C

$$\begin{aligned}
R1 &= Y(I(2)) + Y(I(9)) \\
R2 &= Y(I(2)) - Y(I(9)) \\
R3 &= Y(I(3)) + Y(I(8)) \\
R4 &= Y(I(3)) - Y(I(8)) \\
R5 &= Y(I(4)) + Y(I(7)) \\
R6 &= (Y(I(4)) - Y(I(7))) * C31 \\
R7 &= Y(I(5)) + Y(I(6)) \\
R8 &= Y(I(5)) - Y(I(6)) \\
T0 &= Y(I(1)) + R5 \\
T9 &= Y(I(1)) + R5 * C95 \\
R5 &= R1 + R3 + R7
\end{aligned}$$

$$\begin{aligned}
Y(I(1)) &= T0 + R5 \\
R5 &= T0 + R5 * C95 \\
T0 &= (R3 - R7) * C92 \\
R7 &= (R1 - R7) * C93 \\
R3 &= (R1 - R3) * C94 \\
R1 &= T9 + T0 + R3 \\
T0 &= T9 - T0 - R7 \\
R7 &= T9 + R7 - R3 \\
R9 &= (R2 - R4 + R8) * C31 \\
R3 &= (R4 + R8) * C96 \\
R8 &= (R2 - R8) * C97 \\
R4 &= (R2 + R4) * C98 \\
R2 &= R6 + R3 + R4 \\
R3 &= R6 - R8 - R3 \\
R8 &= R6 + R8 - R4
\end{aligned}$$

C

$$\begin{aligned}
X(IP(2)) &= T1 - R2 \\
X(IP(9)) &= T1 + R2 \\
Y(IP(2)) &= R1 + T2 \\
Y(IP(9)) &= R1 - T2 \\
X(IP(3)) &= T3 + R3 \\
X(IP(8)) &= T3 - R3 \\
Y(IP(3)) &= T0 - T4 \\
Y(IP(8)) &= T0 + T4 \\
X(IP(4)) &= T5 - R9 \\
X(IP(7)) &= T5 + R9 \\
Y(IP(4)) &= R5 + T6 \\
Y(IP(7)) &= R5 - T6 \\
X(IP(5)) &= T7 - R8 \\
X(IP(6)) &= T7 + R8 \\
Y(IP(5)) &= R7 + T8 \\
Y(IP(6)) &= R7 - T8
\end{aligned}$$

C

GOTO 20

C-----WFTA N=16-----

-

C

```
116    R1 = X(I(1)) + X(I(9))
      R2 = X(I(1)) - X(I(9))
      R3 = X(I(2)) + X(I(10))
      R4 = X(I(2)) - X(I(10))
      R5 = X(I(3)) + X(I(11))
      R6 = X(I(3)) - X(I(11))
      R7 = X(I(4)) + X(I(12))
      R8 = X(I(4)) - X(I(12))
      R9 = X(I(5)) + X(I(13))
      R10= X(I(5)) - X(I(13))
      R11 = X(I(6)) + X(I(14))
      R12 = X(I(6)) - X(I(14))
      R13 = X(I(7)) + X(I(15))
      R14 = X(I(7)) - X(I(15))
      R15 = X(I(8)) + X(I(16))
      R16 = X(I(8)) - X(I(16))
      T1 = R1 + R9
      T2 = R1 - R9
      T3 = R3 + R11
      T4 = R3 - R11
      T5 = R5 + R13
      T6 = R5 - R13
      T7 = R7 + R15
      T8 = R7 - R15
      R1 = T1 + T5
      R3 = T1 - T5
      R5 = T3 + T7
      R7 = T3 - T7
      X(IP( 1)) = R1 + R5
      X(IP( 9)) = R1 - R5
      T1 = C81 * (T4 + T8)
      T5 = C81 * (T4 - T8)
      R9 = T2 + T5
      R11= T2 - T5
      R13 = T6 + T1
      R15 = T6 - T1
```

```

T1 = R4 + R16
T2 = R4 - R16
T3 = C81 * (R6 + R14)
T4 = C81 * (R6 - R14)
T5 = R8 + R12
T6 = R8 - R12
T7 = C162 * (T2 - T6)
T2 = C163 * T2 - T7
T6 = C164 * T6 - T7
T7 = R2 + T4
T8 = R2 - T4
R2 = T7 + T2
R4 = T7 - T2
R6 = T8 + T6
R8 = T8 - T6
T7 = C165 * (T1 + T5)
T2 = T7 - C164 * T1
T4 = T7 - C163 * T5
T6 = R10 + T3
T8 = R10 - T3
R10 = T6 + T2
R12 = T6 - T2
R14 = T8 + T4
R16 = T8 - T4
R1 = Y(I(1)) + Y(I(9))
S2 = Y(I(1)) - Y(I(9))
S3 = Y(I(2)) + Y(I(10))
S4 = Y(I(2)) - Y(I(10))
R5 = Y(I(3)) + Y(I(11))
S6 = Y(I(3)) - Y(I(11))
S7 = Y(I(4)) + Y(I(12))
S8 = Y(I(4)) - Y(I(12))
S9 = Y(I(5)) + Y(I(13))
S10= Y(I(5)) - Y(I(13))
S11 = Y(I(6)) + Y(I(14))
S12 = Y(I(6)) - Y(I(14))
S13 = Y(I(7)) + Y(I(15))

```

$$\begin{aligned}
S14 &= Y(I(7)) - Y(I(15)) \\
S15 &= Y(I(8)) + Y(I(16)) \\
S16 &= Y(I(8)) - Y(I(16)) \\
T1 &= R1 + S9 \\
T2 &= R1 - S9 \\
T3 &= S3 + S11 \\
T4 &= S3 - S11 \\
T5 &= R5 + S13 \\
T6 &= R5 - S13 \\
T7 &= S7 + S15 \\
T8 &= S7 - S15 \\
R1 &= T1 + T5 \\
S3 &= T1 - T5 \\
R5 &= T3 + T7 \\
S7 &= T3 - T7 \\
Y(IP(1)) &= R1 + R5 \\
Y(IP(9)) &= R1 - R5 \\
X(IP(5)) &= R3 + S7 \\
X(IP(13)) &= R3 - S7 \\
Y(IP(5)) &= S3 - R7 \\
Y(IP(13)) &= S3 + R7 \\
T1 &= C81 * (T4 + T8) \\
T5 &= C81 * (T4 - T8) \\
S9 &= T2 + T5 \\
S11 &= T2 - T5 \\
S13 &= T6 + T1 \\
S15 &= T6 - T1 \\
T1 &= S4 + S16 \\
T2 &= S4 - S16 \\
T3 &= C81 * (S6 + S14) \\
T4 &= C81 * (S6 - S14) \\
T5 &= S8 + S12 \\
T6 &= S8 - S12 \\
T7 &= C162 * (T2 - T6) \\
T2 &= C163 * T2 - T7 \\
T6 &= C164 * T6 - T7 \\
T7 &= S2 + T4
\end{aligned}$$

$$\begin{aligned}
T8 &= S2 - T4 \\
S2 &= T7 + T2 \\
S4 &= T7 - T2 \\
S6 &= T8 + T6 \\
S8 &= T8 - T6 \\
T7 &= C165 * (T1 + T5) \\
T2 &= T7 - C164 * T1 \\
T4 &= T7 - C163 * T5 \\
T6 &= S10 + T3 \\
T8 &= S10 - T3 \\
S10 &= T6 + T2 \\
S12 &= T6 - T2 \\
S14 &= T8 + T4 \\
S16 &= T8 - T4 \\
X(IP(2)) &= R2 + S10 \\
X(IP(16)) &= R2 - S10 \\
Y(IP(2)) &= S2 - R10 \\
Y(IP(16)) &= S2 + R10 \\
X(IP(3)) &= R9 + S13 \\
X(IP(15)) &= R9 - S13 \\
Y(IP(3)) &= S9 - R13 \\
Y(IP(15)) &= S9 + R13 \\
X(IP(4)) &= R8 - S16 \\
X(IP(14)) &= R8 + S16 \\
Y(IP(4)) &= S8 + R16 \\
Y(IP(14)) &= S8 - R16 \\
X(IP(6)) &= R6 + S14 \\
X(IP(12)) &= R6 - S14 \\
Y(IP(6)) &= S6 - R14 \\
Y(IP(12)) &= S6 + R14 \\
X(IP(7)) &= R11 - S15 \\
X(IP(11)) &= R11 + S15 \\
Y(IP(7)) &= S11 + R15 \\
Y(IP(11)) &= S11 - R15 \\
X(IP(8)) &= R4 - S12 \\
X(IP(10)) &= R4 + S12 \\
Y(IP(8)) &= S4 + R12
\end{aligned}$$

```
      Y(IP(10)) = S4 - R12
C
      GOTO 20
C
20      CONTINUE
10      CONTINUE
      RETURN
      END
```

Appendix 4: Programs for Short FFTs

This appendix will discuss efficient short FFT programs that can be used in both the [Cooley-Tukey](#) and the [Prime Factor FFT algorithms](#). Links and references are given to Fortran listings that can be used "as is" or put into the indexed loops of existing programs to give greater efficiency and/or a greater variety of allowed lengths. Special programs have been written for lengths: $N = 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19, 25$, etc.

In the early days of the FFT, multiplication was done in software and was, therefore, much slower than an addition. With modern hardware, a floating point multiplication can be done in one clock cycle of the computer, microprocessor, or DSP chip, requiring the same time as an addition. Indeed, in some computers and many DSP chips, both a multiplication and an addition (or accumulation) can be done in one cycle while the indexing and memory access is done in parallel. Most of the algorithms described here are not hardware architecture specific but are designed to minimize both multiplications and additions.

The most basic and often used length FFT (or DFT) is for $N = 2$. In the Cooley Tukey FFT, it is called a "butterfly" and its reason for fame is requiring no multiplications at all, only one complex addition and one complex subtraction and needing only one complex temporary storage location. This is illustrated in [Figure 1: The Prime Factor and Winograd Transform Algorithms](#) and code is shown in [Figure 2: The Prime Factor and Winograd Transform Algorithms](#). The second most used length is $N = 4$ because it is the only other short length requiring no multiplications and a minimum of additions. All other short FFT require some multiplication but for powers of two, $N = 8$ and $N = 16$ require few enough to be worth special coding for some situations.

Code for other short lengths such as the primes $N = 3, 5, 7, 11, 13, 17$, and 19 and the composites $N = 9$ and 25 are included in the programs for the prime factor algorithm or the WFTA. They are derived using the theory in Chapters 5, 6, and 9. They can also be found in references ... and

If these short FFTs are used as modules in the basic prime factor algorithm (PFA), then the straight forward development used for the modules in Figure 17.12 are used. However if the more complicated indexing use to achieve in-order, in-place calculation used in {xxxxx} require different code.

For each of the indicated lengths, the computer code is given in a Connexions module.

They are not in the collection [Fast Fourier Transforms](#) as the printed version would be too long. However, one can link to them on-line from the following buttons: [N=2](#) [N=3](#) [N=4](#) [N=5](#) [N=7](#) [N= 8](#) [N= 9](#) [N= 11](#) [N= 13](#) [N= 16](#) [N= 17](#) [N= 19](#) [N= 25](#) Versions for the in-place, in-order prime factor algorithm {pfa} can be obtained from: [N=2](#) [N=3](#) [N=4](#) [N=5](#) [N=7](#) [N=8](#) [N=9](#) [N=11](#) [N=13](#) [N=16](#) [N=17](#) [N=19](#) [N=25](#) A technical report that describes the length 11, 13, 17, and 19 is in {report 8105} and another technical report that describes a program that will automatically generate a prime length FFT and its flow graph si in {report xxx}.